Essential Guide to Seaborn

Visual storytelling made simple.

Master statistical data visualization with seaborn. Create informative and beautiful plots directly from your data

rubió
Metabolomics

Ibon Martínez-Arranz | imartinez@labrubiocom
Data Science Manager at Rubió Metabolomics
www.rubiometabolomics.com

Ibon Martínez-Arranz, PhD in Mathematics and Statistics, holds MSc degrees in Applied Statistical Techniques and Mathematical Modeling. He has extensive expertise in statistical modeling and advanced data analysis techniques. Since 2017, he has led the Data Science area at Rubió Metabolomics, driving predictive model development and statistical computing management for metabolomics, data handling, and R&D projects. His doctoral research in Mathematics investigated and adapted genetic algorithms to improve the classification of NAFLD subtypes.

Itziar Mincholé Canals | iminchole@labrubio.com
Data Specialist at Rubió Metabolomics
www.rubiometabolomics.com

Itziar Mincholé Canals has a degree in physical sciences (1999) from the University of Zaragoza (Spain). In 1999 she began developing her professional career as a software analyst and developer, mainly working on web development projects and database programming in different sectors. In 2009 she joined OWL Metabolomics as a bioinformatician, where she has participated in several R&D projects and has developed several software tools for metabolomics. In 2016 she completed the master's degree in applied statistics with R software. After joining the Data Science department, she also supports data analysis.

Essential Guide to Seaborn

Ibon Martínez-Arranz

Life
Feels
Good

rubió
Metabolomics

Contents

Essential Guide to Seaborn

Introduction and Setup

Why Seaborn?

Seaborn is a Python data visualization library based on Matplotlib. It provides a high-level, user-friendly API for creating informative and attractive statistical graphics. While Matplotlib is highly flexible and powerful, Seaborn offers a set of tools that make common visualization tasks faster and easier, especially when working with pandas DataFrames.

Why should you use Seaborn?

- Easy integration with pandas DataFrames.
- Beautiful default styles.
- Built-in functions for common statistical plots.
- Simplified syntax compared to Matplotlib.
- Easy customization and extension with Matplotlib.

This book will guide you through the main functionalities of Seaborn, starting from simple plots and progressing to more advanced topics like grids, regression plots, and case studies.

Installation and Requirements

Before starting, you need to install Seaborn and its dependencies. We recommend using a virtual environment.

```
pip install seaborn pandas matplotlib
```

We will also use `jupyter` to execute and display plots inside notebooks:

```
pip install jupyter
```

Recommended imports:

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

sns.set_theme()  # Optional but recommended
```

Your First Plot

Let's create our first simple plot to make sure everything works:

```
# Create some dummy data
data = pd.DataFrame({
    "x": range(10),
    "y": [i ** 2 for i in range(10)]
})

sns.lineplot(data=data, x="x", y="y")
plt.show()
```

You should see a simple line plot.

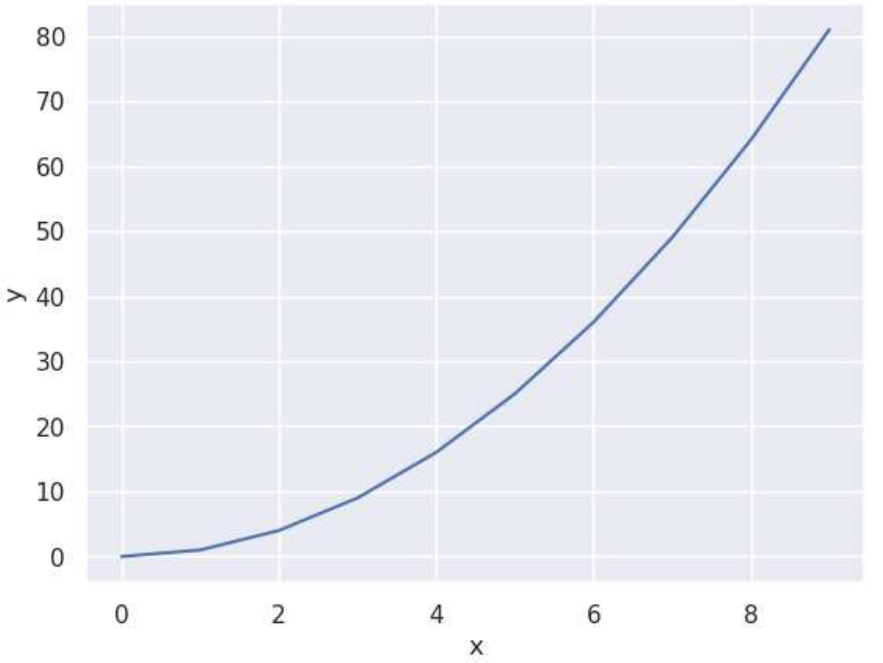

Figure 1: Image generated by the provided code.

Datasets We Will Use

Throughout this book, we will work with several datasets:

Penguins dataset

The `penguins` dataset is a modern alternative to the classic iris dataset. It provides measurements of three penguin species collected from islands in Palmer Archipelago, Antarctica.

- Variables: species, island, bill length, bill depth, flipper length, body mass, sex, year.

Tips dataset

A classic dataset about restaurant bills and tips, including total bill, tip, gender of the customer, smoking status, day of the week, time of the day, and party size.

- Variables: total_bill, tip, sex, smoker, day, time, size.

Flights dataset

A time series dataset showing the number of passengers over years and months. Very useful for line plots and heatmaps.

- Variables: year, month, passengers.

Example: First Look at Penguins Dataset

```
# Load penguins dataset
penguins = sns.load_dataset("penguins")

# Display the first rows
penguins.head()
```

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex
0	Adelie	Torgersen	39.1	18.7	181	3750	Male
1	Adelie	Torgersen	39.5	17.4	186	3800	Female
2	Adelie	Torgersen	40.3	18	195	3250	Female

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex
3	Adelie	Torgersen	nan	nan	nan	nan	nan
4	Adelie	Torgersen	36.7	19.3	193	3450	Female

Example: First Look at Tips Dataset

```
# Load tips dataset
tips = sns.load_dataset("tips")

# Display the first rows
tips.head()
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.5	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

Example: First Look at Flights Dataset

```
# Load tips dataset
flights = sns.load_dataset("flights")

# Display the first rows
flights.head()
```

	year	month	passengers
0	1949	Jan	112
1	1949	Feb	118
2	1949	Mar	132
3	1949	Apr	129
4	1949	May	121

In the next chapter, we will begin exploring **relational plots**, the backbone of scatter plots, line plots, and more.

Relational Plots

Introduction

Relational plots are used to display relationships between two or more variables. This is one of the most common tasks in data visualization, especially when performing exploratory data analysis (EDA).

Seaborn provides two main functions for relational plots: - `scatterplot()` for point-based visualizations. - `lineplot()` for trend or time series visualizations.

In this chapter, we will learn how to use them step by step, from the simplest usage to advanced customizations using arguments such as `hue`, `size`, and `style`. These arguments are especially useful when we want to encode additional information visually.

Scatter Plots

The scatter plot is used to visualize the relationship between two numeric variables. Let's start with a simple example using the `penguins` dataset.

```
# Load dataset
penguins = sns.load_dataset("penguins")

# Basic scatter plot
```

```
sns.scatterplot(data=penguins,
                x="bill_length_mm",
                y="bill_depth_mm")
plt.show()
```

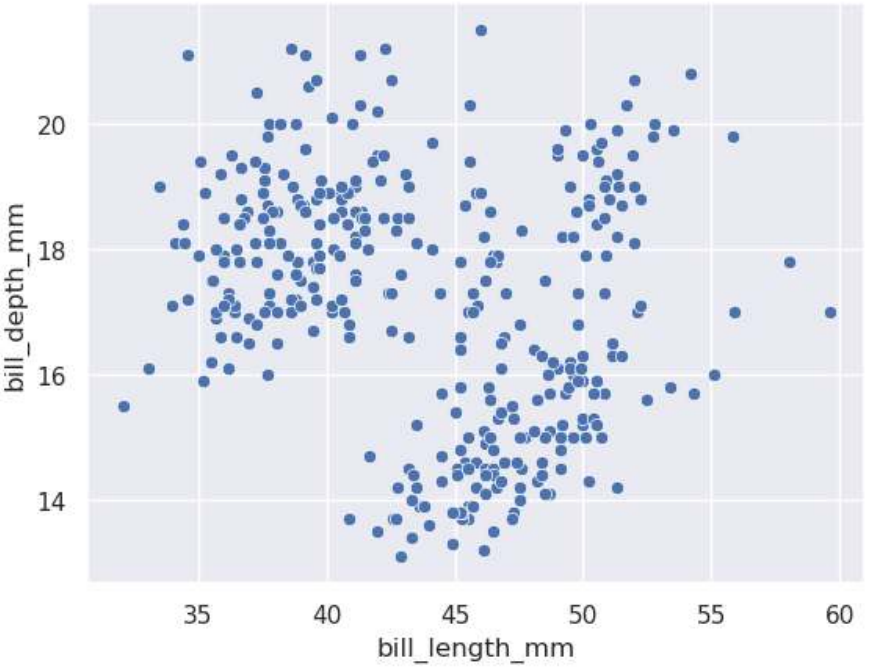

Figure 1: Image generated by the provided code.

Explanation:

Each point represents one penguin. The x axis shows the bill length and the y axis the bill depth.

Adding color with hue

Seaborn allows you to easily map a categorical variable to the color of the points using the hue argument.

```
sns.scatterplot(data=penguins,
                x="bill_length_mm",
                y="bill_depth_mm",
                hue="species")
plt.show()
```

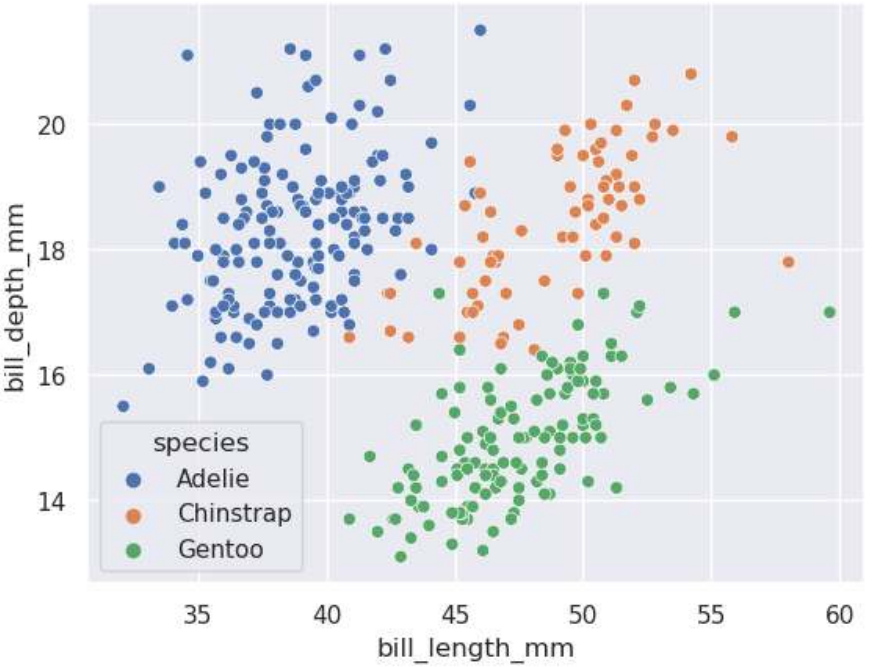

Figure 2: Image generated by the provided code.

Explanation:

Now, each species has a different color. This makes it easier to spot patterns by species.

Adding size and style

You can add even more information by using the `size` and `style` arguments.

```
sns.scatterplot(
    data=penguins,
    x="bill_length_mm",
    y="bill_depth_mm",
    hue="species",
    size="flipper_length_mm",
    style="sex"
)
plt.show()
```

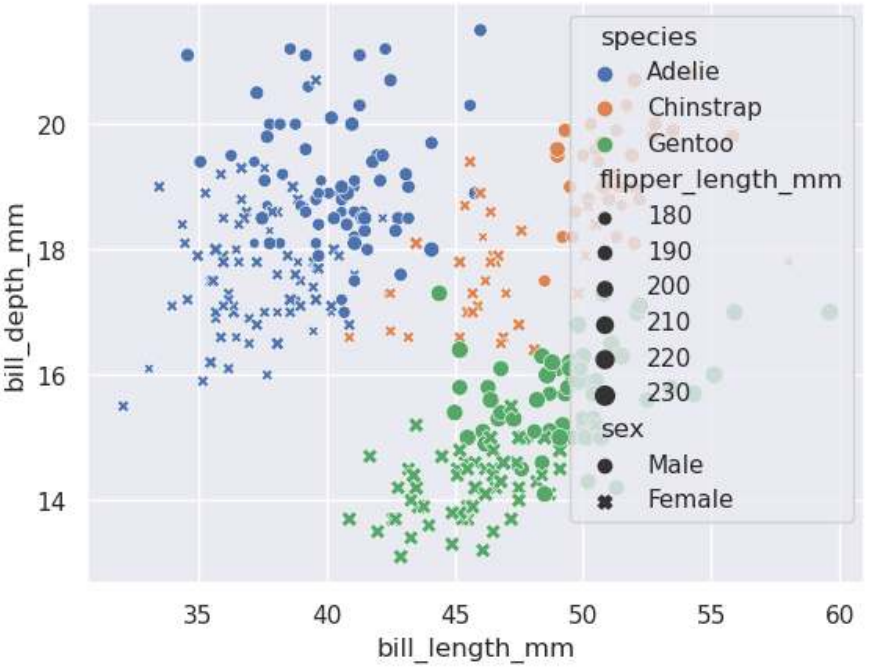

Figure 3: Image generated by the provided code.

Explanation:

- The size of the points represents the flipper length.
- The style (shape) of the points indicates the sex of each penguin.

Notice how Seaborn automatically handles legends when you use multiple encodings.

Line Plots

Line plots are mainly used to show trends over a continuous variable, typically time or ordered data.

Let's move to the `flights` dataset, which is perfect for this type of visualization.

```
flights = sns.load_dataset("flights")
flights.head()
```

	year	month	passengers
0	1949	Jan	112
1	1949	Feb	118
2	1949	Mar	132
3	1949	Apr	129
4	1949	May	121

Basic line plot

```
sns.lineplot(data=flights, x="year", y="passengers")
plt.show()
```

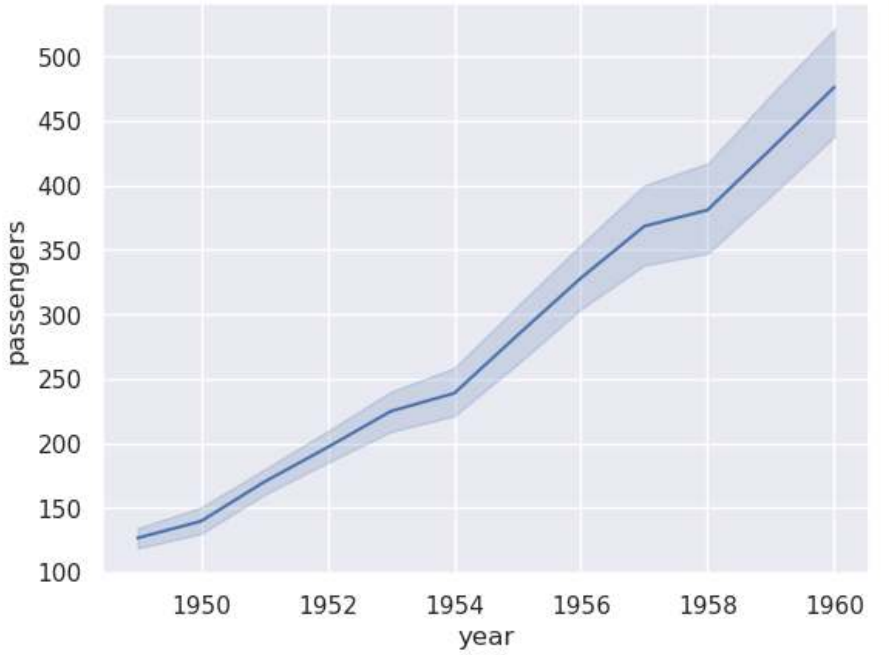

Figure 4: Image generated by the provided code.

Explanation:

This plot shows the average number of passengers for each year, aggregating all months.

Grouped line plot with hue

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

```
4                 hue="month")
5 plt.show()
```

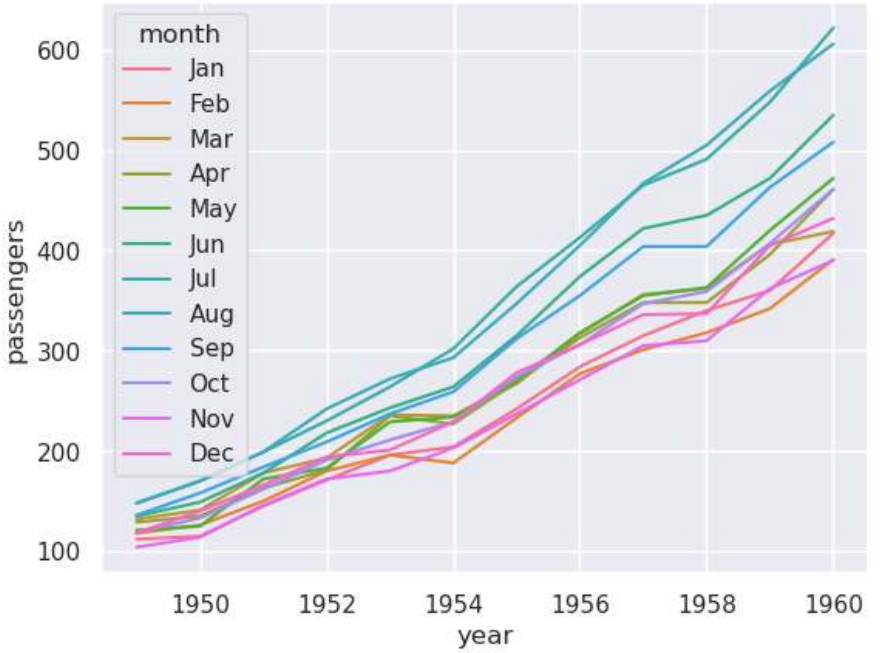

Figure 5: Image generated by the provided code.

Explanation:

Now we are splitting the data by `month`. Each line represents a month, making it possible to analyze seasonality and trends over years.

Advanced Customizations

Seaborn makes it easy to combine multiple encodings:

Line styles and markers

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

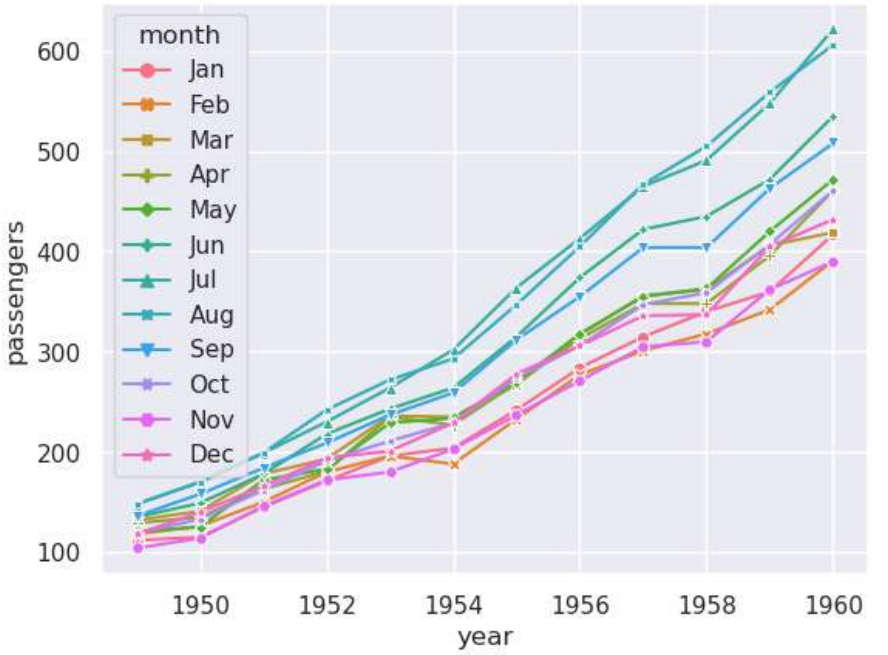

Figure 6: Image generated by the provided code.

Explanation:

- Different months now have both different colors and different line styles (markers).
- This is especially helpful when creating plots for print, where color alone might not be sufficient.

Relational Plot Summary

Argument	Description
`hue`	Maps a variable to color
`size`	Maps a variable to marker size
`style`	Maps a variable to marker style (scatter) or line style (lineplot)
`markers`	Adds markers to lineplot
`dashes`	Controls whether to use dashed lines

When to use scatter vs. line plots?

- Use **scatterplot** when your `x` variable is **not ordered** (or when showing individual points is relevant).
- Use **lineplot** when your `x` variable is **ordered** or **continuous**, especially in time series.
- Don't hesitate to combine both when you need to emphasize both trend and individual data points.

In the next chapter, we will dive into **distribution plots**, which are essential for understanding the shape of your data and identifying patterns such as skewness, modality, or outliers.

Distribution Plots

Introduction

Distribution plots are essential for understanding the structure of a single variable. They help you to:

- Check the distribution shape (normal, skewed, bimodal).
- Detect outliers.
- Compare distributions between groups.

Seaborn provides several powerful functions to visualize distributions:

- `histplot()` for histograms.
- `kdeplot()` for kernel density estimation (KDE).
- `ecdfplot()` for empirical cumulative distribution functions.
- `rugplot()` for marginal ticks.

In this chapter, we will explore each of these functions and learn how to combine them.

Histograms with `histplot()`

Histograms are one of the most common tools to understand a variable's distribution.

Basic Histogram

```
1 sns.histplot(data=penguins, x="flipper_length_mm")
2 plt.show()
```

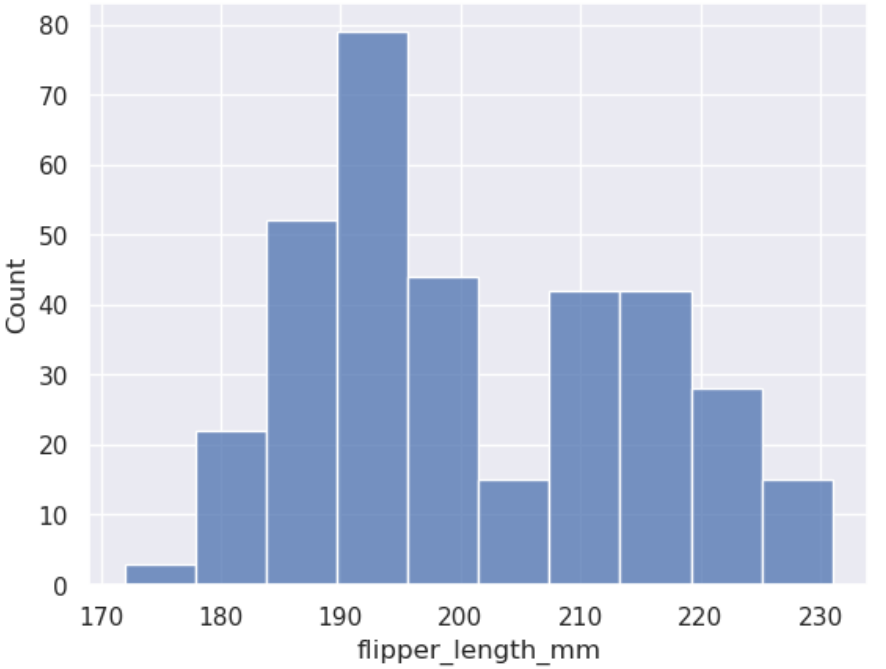

Figure 1: Image generated by the provided code.

Explanation:

The histogram counts how many penguins fall into each bin of `flipper_length_mm`. By default, Seaborn automatically chooses the number of bins.

Customizing bins

```
sns.histplot(data=penguins, x="flipper_length_mm", bins
    =30)
plt.show()
```

Changing the number of bins allows you to control the granularity of the histogram.

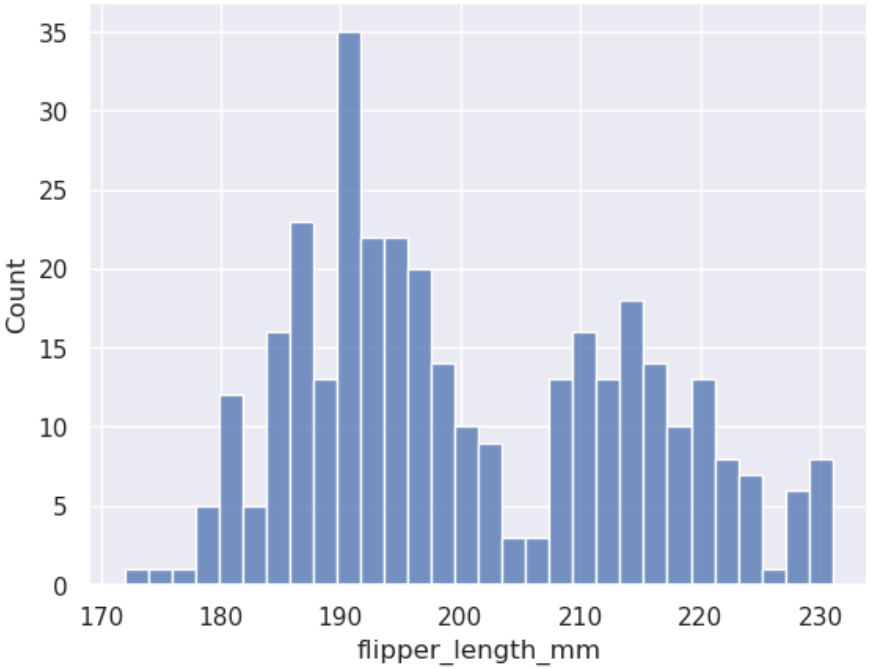

Figure 2: Image generated by the provided code.

Histogram by category using hue

```
sns.histplot(data=penguins,
             x="flipper_length_mm",
             hue="species",
             element="step")
plt.show()
```

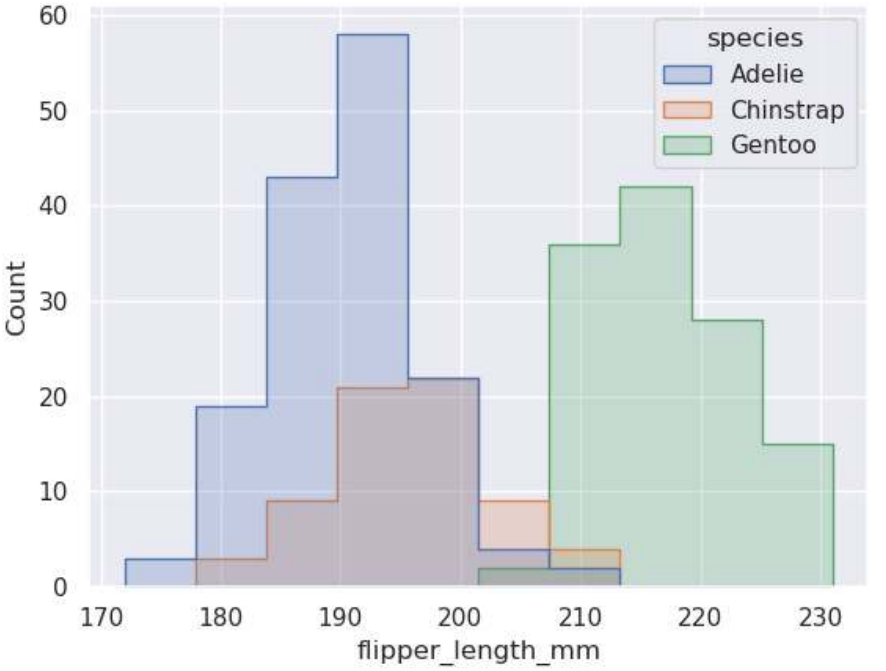

Figure 3: Image generated by the provided code.

Explanation:

- The `hue` argument allows you to separate the distribution by species.
- The `element="step"` makes the plot less cluttered by drawing outlined

histograms.

Kernel Density Estimation with `kdeplot()`

KDE plots are smoothed versions of histograms that help to see the distribution shape more clearly.

Basic KDE

```
sns.kdeplot(data=penguins, x="flipper_length_mm")
plt.show()
```

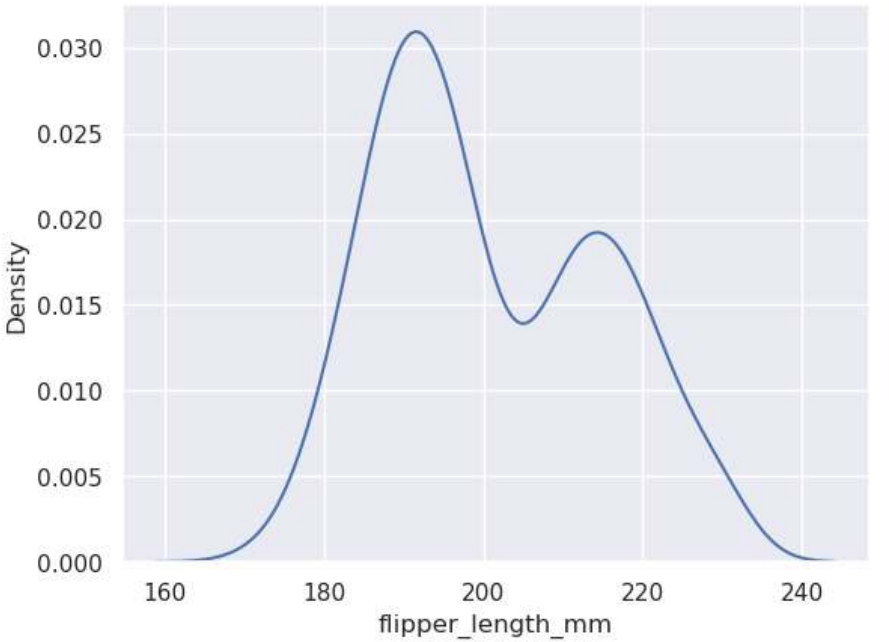

Figure 4: Image generated by the provided code.

Explanation:

This shows a smoothed estimate of the probability density function.

KDE by group with hue

```
sns.kdeplot(data=penguins,
            x="flipper_length_mm",
            hue="species",
            fill=True)
plt.show()
```

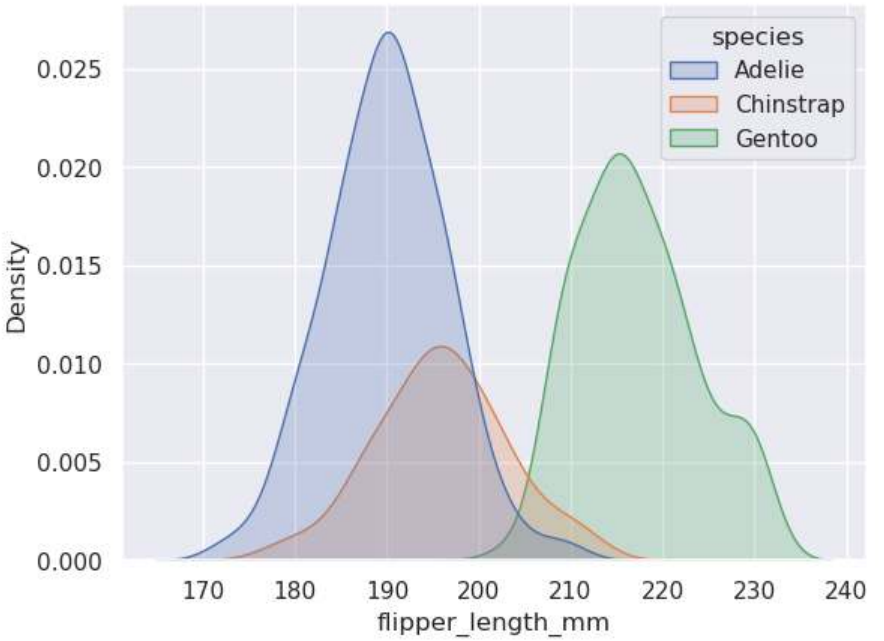

Figure 5: Image generated by the provided code.

Explanation:

- By using `hue`, you can compare the distributions of flipper lengths by species.
- `fill=True` will fill the area under the curves.

Empirical Cumulative Distribution Function with `ecdfplot()`

An ECDF shows the cumulative proportion of the data.

Basic ECDF

```
1 sns.ecdfplot(data=penguins, x="flipper_length_mm")
2 plt.show()
```

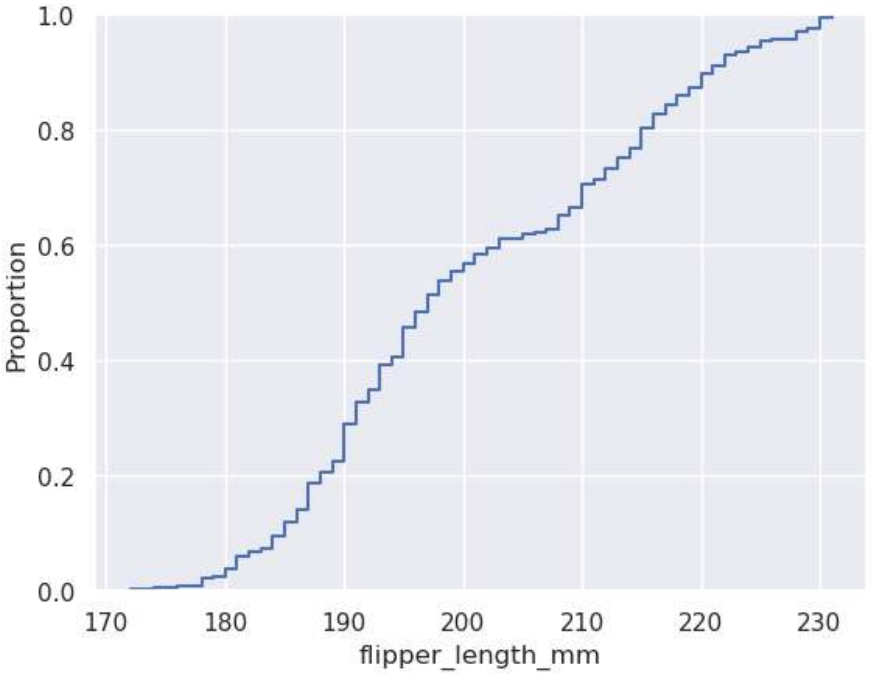

Figure 6: Image generated by the provided code.

Explanation:

For each value of `flipper_length_mm`, the plot shows the proportion of penguins with flipper lengths less than or equal to that value.

ECDF by group

```
sns.ecdfplot(data=penguins,
             x="flipper_length_mm",
             hue="species")
plt.show()
```

This visualization is helpful for comparing distributions across groups.

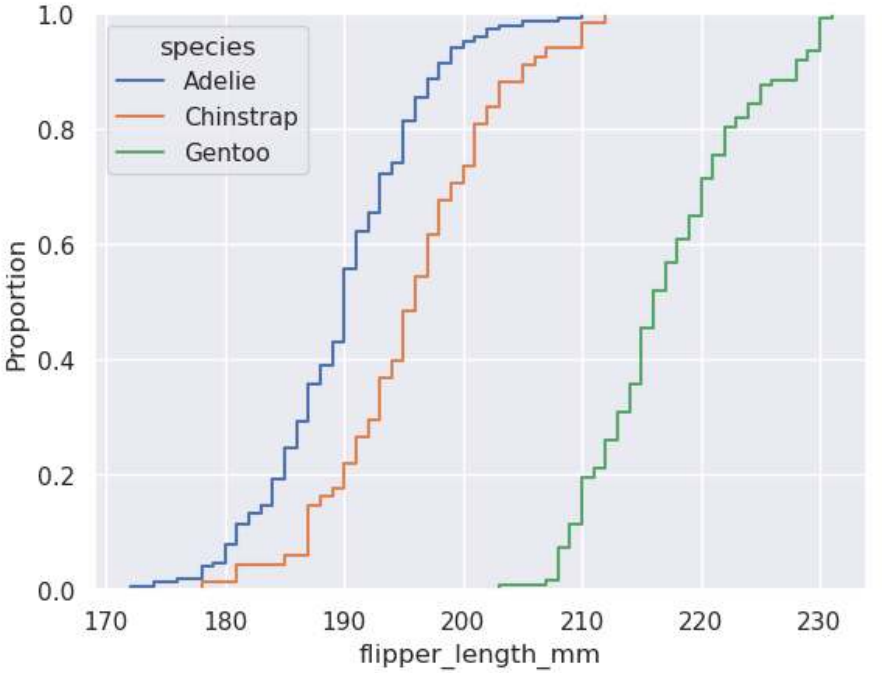

Figure 7: Image generated by the provided code.

Rug Plot with `rugplot()`

A rug plot adds small tick marks along the axis to show the actual data points.

Basic Rug Plot

```
1  sns.rugplot(data=penguins, x="flipper_length_mm")
2  plt.show()
```

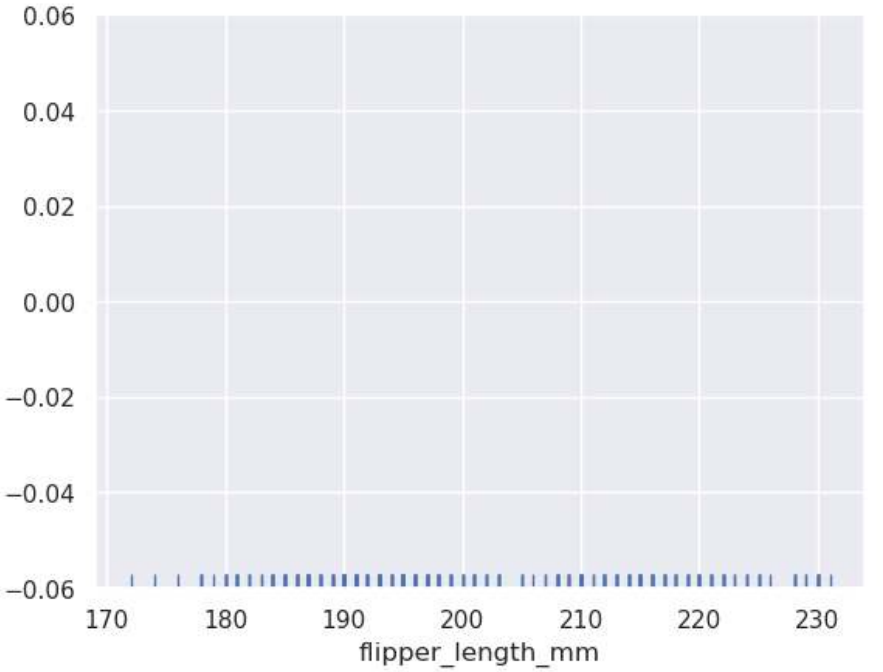

Figure 8: Image generated by the provided code.

Explanation:

Each small vertical line corresponds to a data point. It is useful to visualize the density of data points, especially when combined with other plots.

Combining KDE and Rug Plot

```
1  sns.kdeplot(data=penguins, x="flipper_length_mm")
2  sns.rugplot(data=penguins, x="flipper_length_mm")
3  plt.show()
```

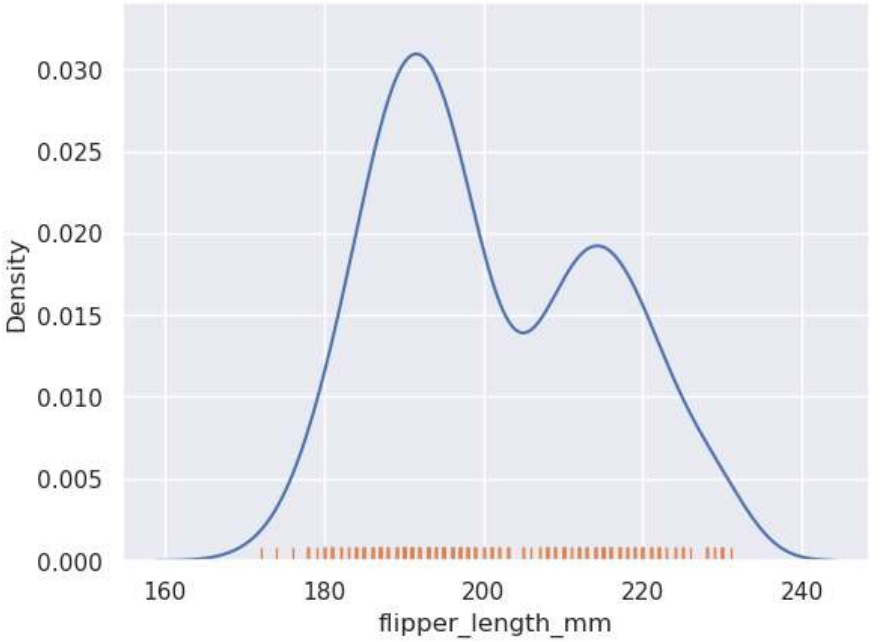

Figure 9: Image generated by the provided code.

Explanation:

The combination of KDE and rug plots allows you to see both the smoothed distribution and the actual data points.

Additional Tips

- `histplot()`, `kdeplot()`, and `ecdfplot()` accept common arguments like `hue`, `multiple`, `element`, `fill`, and `common_norm`.
- You can easily combine distribution plots to create layered visualizations.
- KDE is sensitive to outliers; use it carefully with noisy data.

Summary Table

Plot	Purpose
`histplot()`	Show frequency counts (histogram)
`kdeplot()`	Estimate and plot the distribution's density
`ecdfplot()`	Display cumulative distribution function
`rugplot()`	Show individual data points along an axis

In the next chapter, we will explore **Categorical Plots**, which are essential when working with qualitative data.

Categorical Plots

Introduction

Categorical plots are among the most frequently used types of plots in data analysis. They are essential for visualizing the relationship between categorical and numerical variables or for comparing distributions across different categories.

Seaborn provides multiple functions to create categorical plots, each with its own strengths:

- `barplot()`: Estimate and display the mean (or other estimator) of a numerical variable across categories.
- `countplot()`: Display counts of observations in each categorical bin.
- `boxplot()`: Show distributions with quartiles and outliers.
- `violinplot()`: Combine a boxplot and a KDE for richer distribution information.
- `stripplot()`: Display all individual data points, useful for small datasets.
- `swarmplot()`: Display individual points without overlap.

In this chapter, we will explore these plots in depth.

Bar Plot with `barplot()`

Bar plots are used to show the average of a numerical variable for each category, often with confidence intervals.

Basic bar plot

```
1 sns.barplot(data=tips, x="day", y="total_bill")
2 plt.show()
```

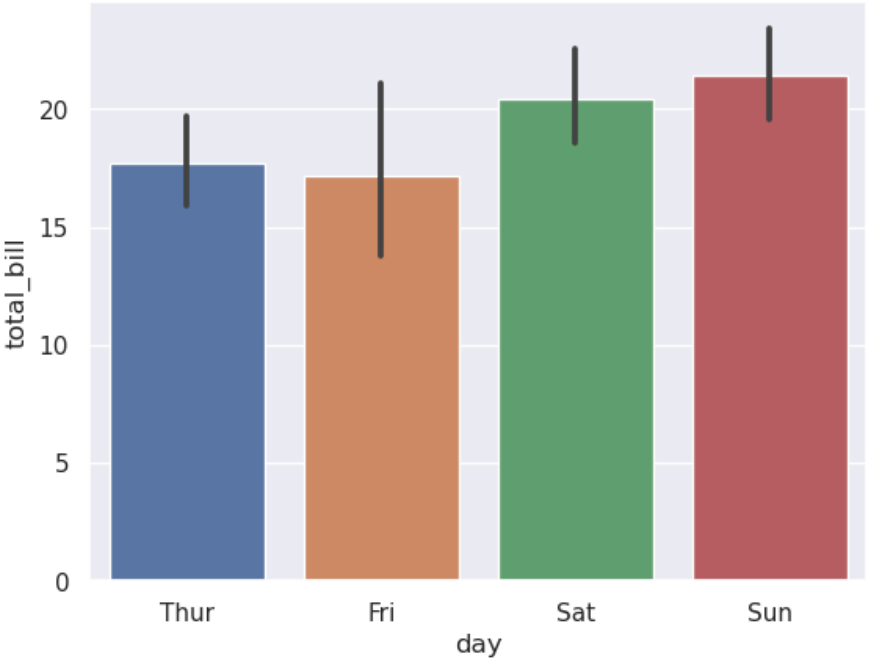

Figure 1: Image generated by the provided code.

Explanation:

This plot shows the **average** `total_bill` for each day of the week.

Custom estimator

You can change the estimator (by default is the mean) to other summary statistics like the median:

```
from numpy import median

sns.barplot(data=tips,
            x="day",
            y="total_bill",
            estimator=median)
plt.show()
```

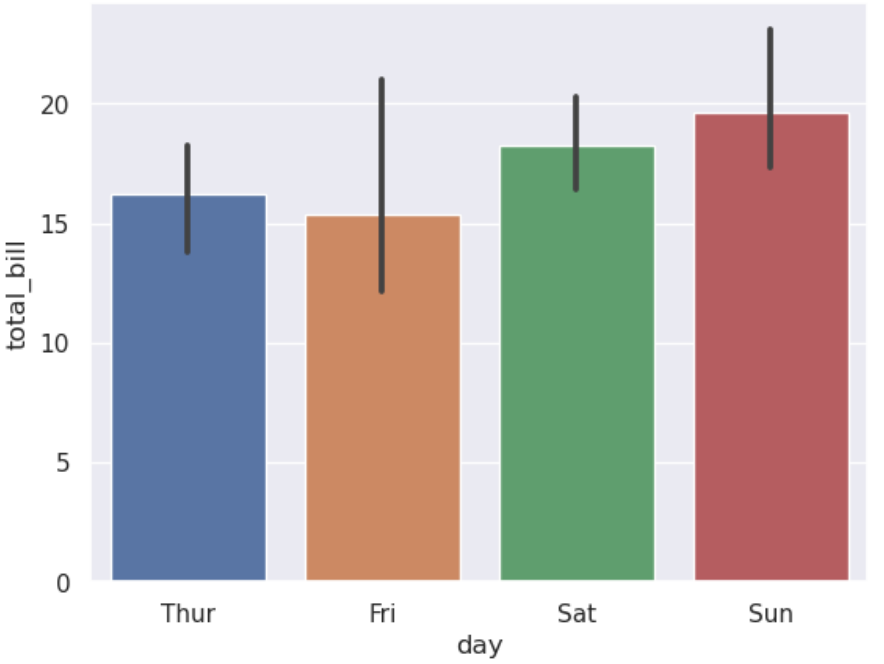

Figure 2: Image generated by the provided code.

Bar plot with hue

```
sns.barplot(data=tips,
            x="day",
            y="total_bill",
            hue="sex")
plt.show()
```

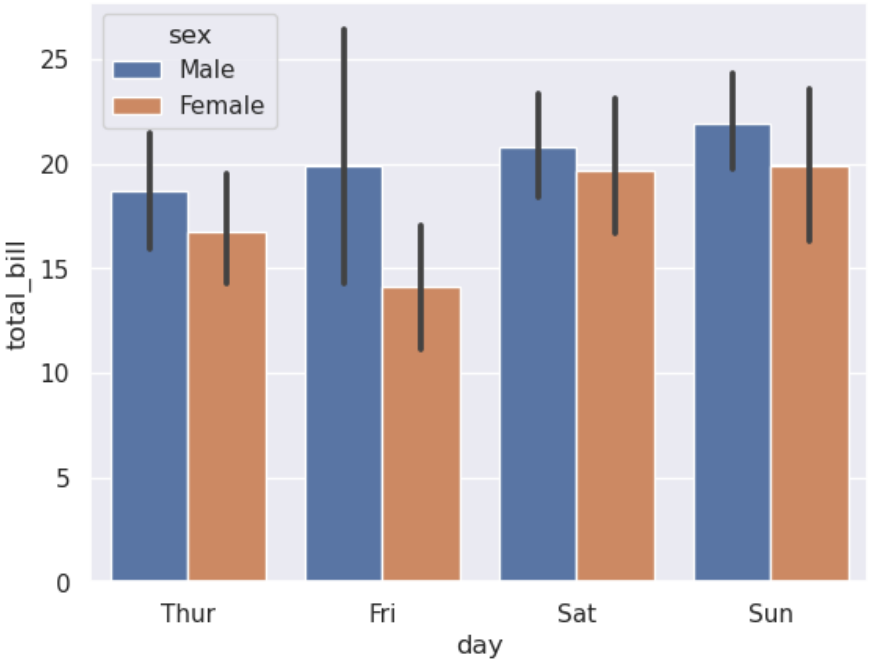

Figure 3: Image generated by the provided code.

Explanation:

The bars are split by `sex` within each day.

Count Plot with `countplot()`

Unlike `barplot()`, `countplot()` directly counts the number of observations in each category.

Basic count plot

```
1  sns.countplot(data=tips, x="day")
2  plt.show()
```

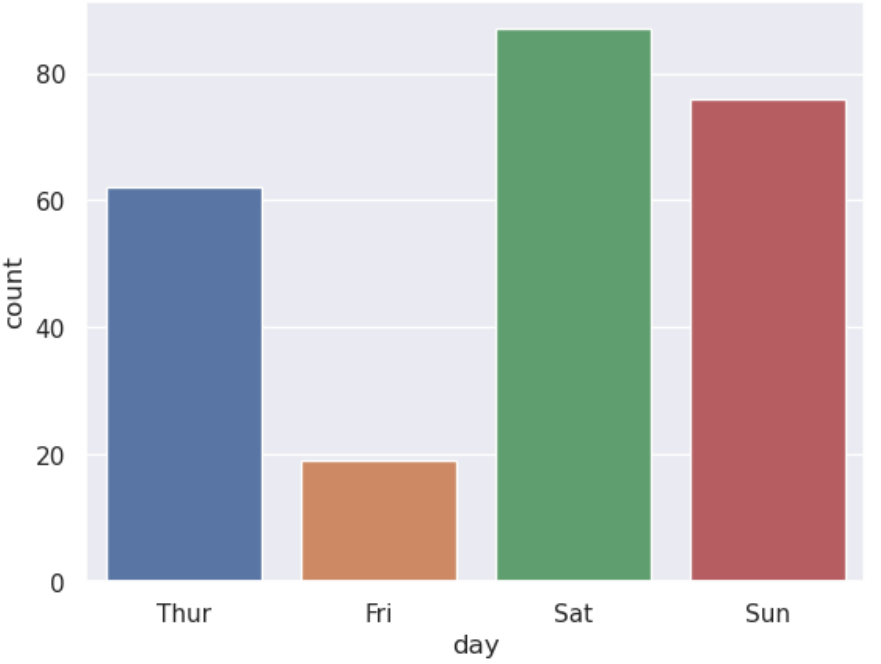

Figure 4: Image generated by the provided code.

Explanation:

Shows how many records there are for each day in the dataset.

Count plot with hue

```
sns.countplot(data=tips, x="day", hue="sex")
plt.show()
```

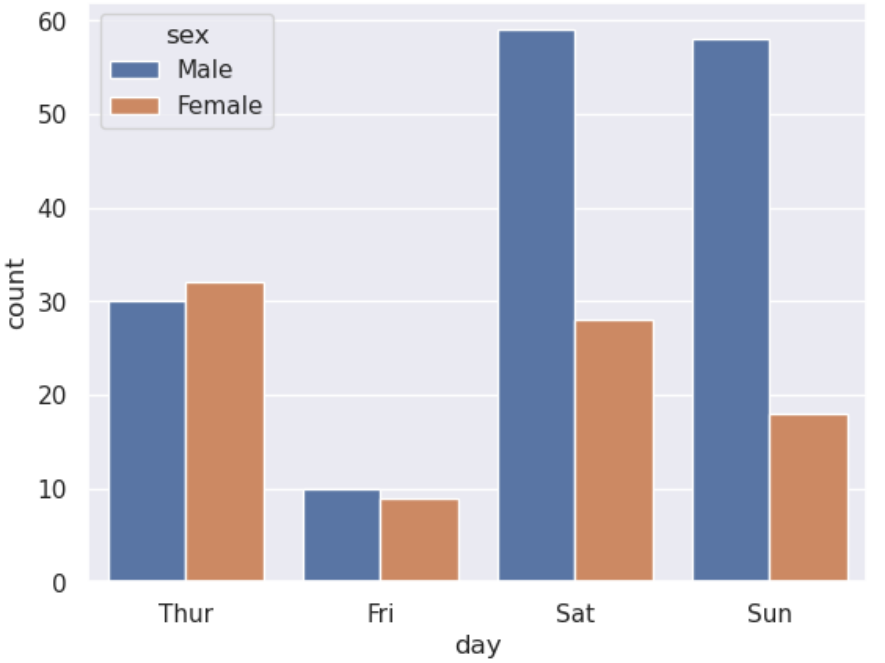

Figure 5: Image generated by the provided code.

This is helpful for visualizing how the counts vary between subgroups.

Box Plot with `boxplot()`

Box plots show the distribution of a numerical variable using quartiles and highlighting potential outliers.

Basic box plot

```
1  sns.boxplot(data=tips, x="day", y="total_bill")
2  plt.show()
```

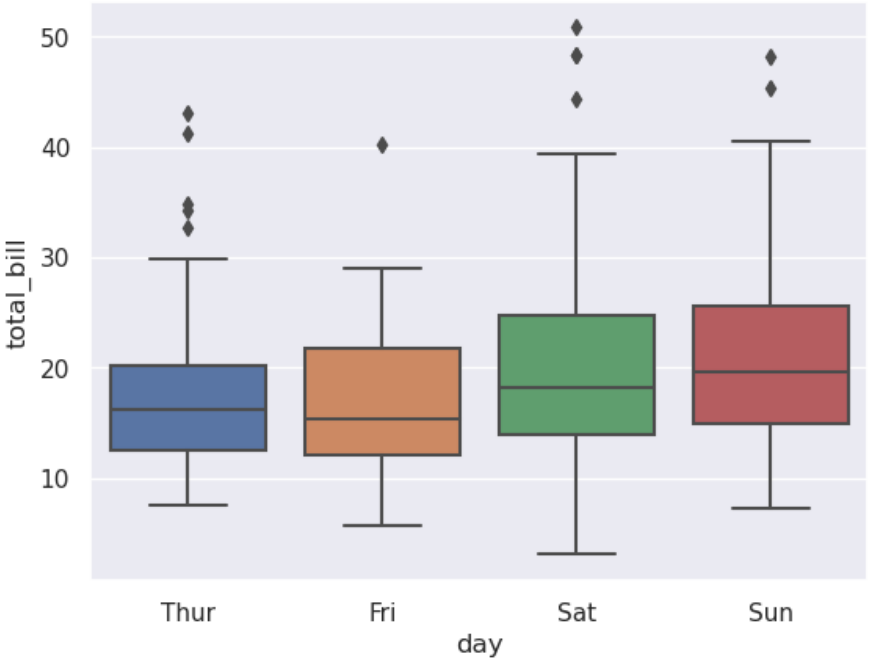

Figure 6: Image generated by the provided code.

Explanation:

- The box represents the interquartile range (IQR).
- The line inside the box shows the median.
- The “whiskers” extend to 1.5 * IQR, and points beyond are considered outliers.

Box plot with hue

```
sns.boxplot(data=tips, x="day", y="total_bill", hue="sex"
    )
plt.show()
```

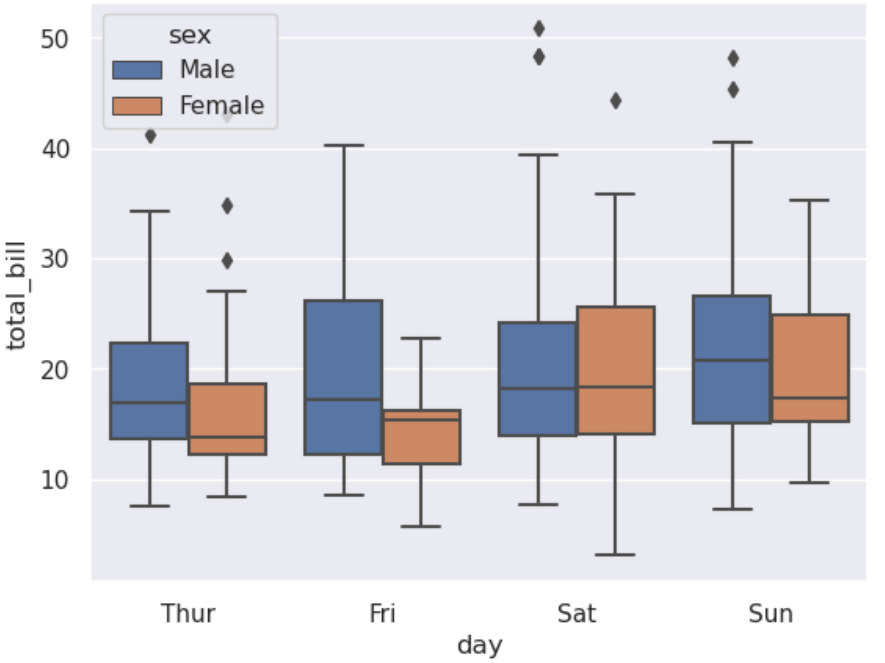

Figure 7: Image generated by the provided code.

Adding hue allows us to compare distributions between subgroups.

Violin Plot with `violinplot()`

Violin plots combine the information of a boxplot and a kernel density estimation (KDE).

Basic violin plot

```
sns.violinplot(data=tips, x="day", y="total_bill")
plt.show()
```

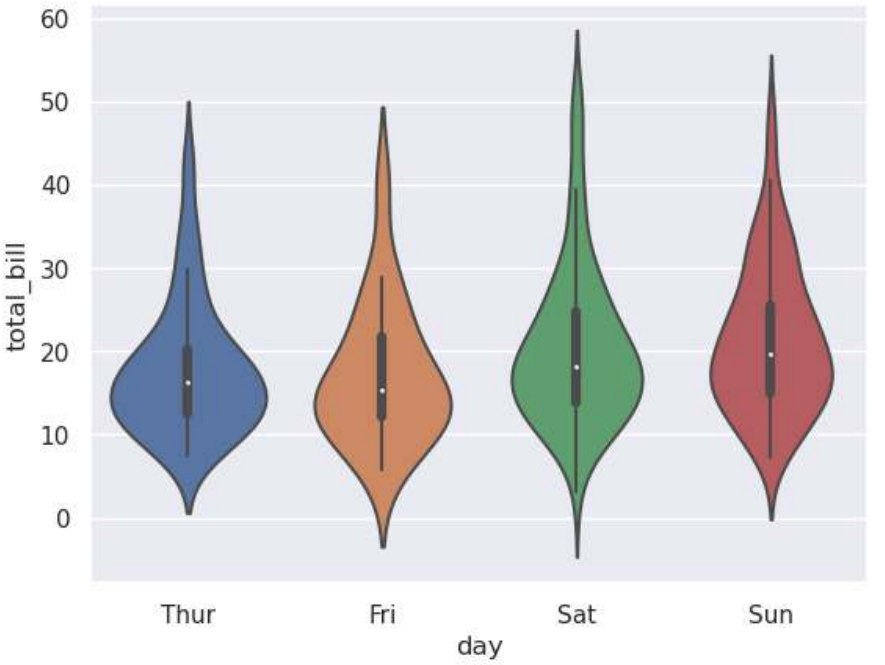

Figure 8: Image generated by the provided code.

Explanation:

You can see the distribution shape and quartiles simultaneously.

Violin plot with split and hue

```
sns.violinplot(data=tips,
               x="day",
               y="total_bill",
               hue="sex",
```

```
                split=True)
plt.show()
```

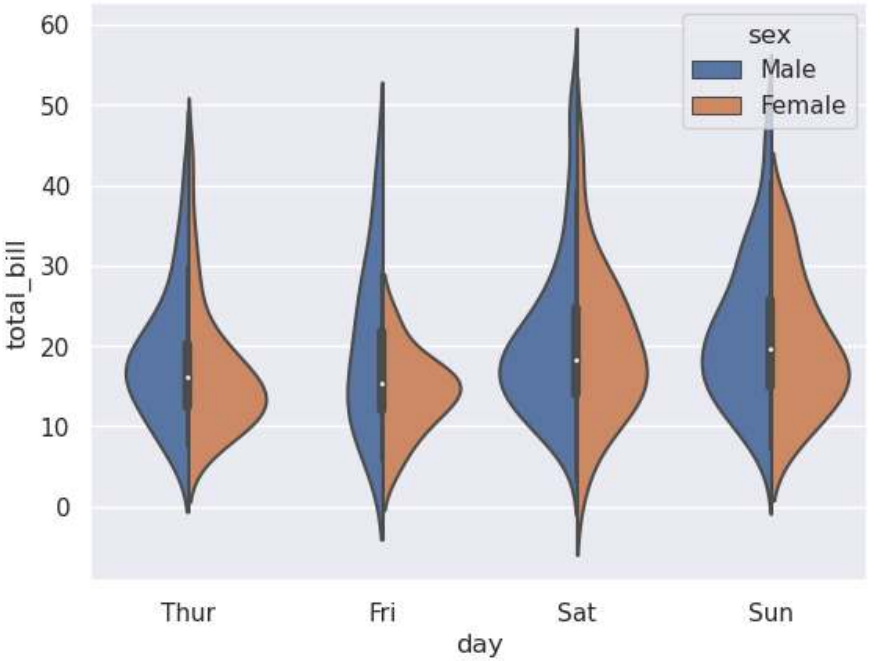

Figure 9: Image generated by the provided code.

This allows you to see both distributions in the same "violin" when `hue` has two categories.

Strip Plot with `stripplot()`

Strip plots are scatter plots where points are arranged along a categorical axis.

Basic strip plot

```
1 sns.stripplot(data=tips, x="day", y="total_bill")
2 plt.show()
```

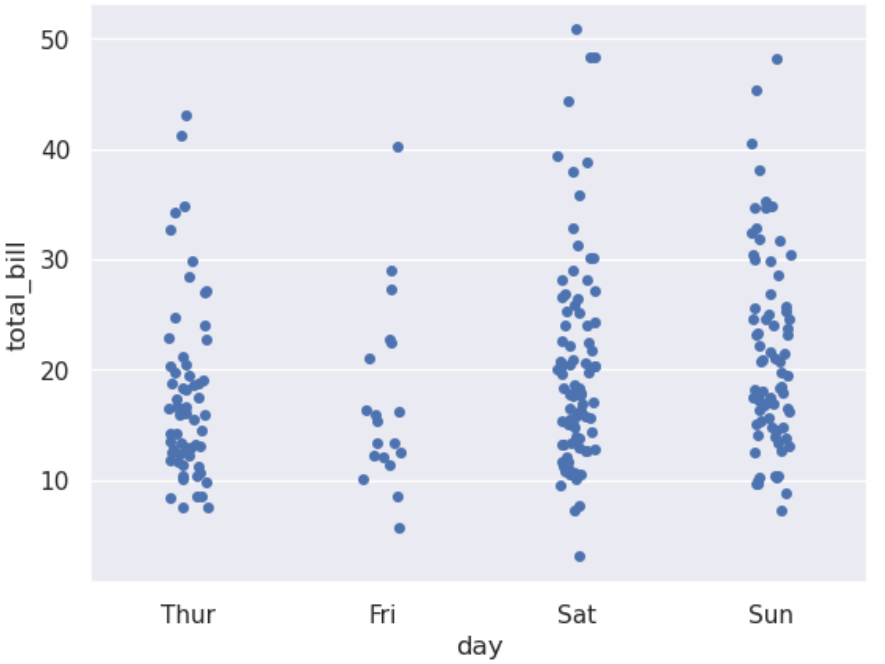

Figure 10: Image generated by the provided code.

Explanation:

Useful to see all data points, especially with small datasets.

Strip plot with jitter and hue

```
sns.stripplot(data=tips,
              x="day",
              y="total_bill",
              jitter=True,
              hue="sex")
plt.show()
```

- `jitter=True` spreads the points to avoid overlap.
- Combining with `hue` helps to distinguish subgroups.

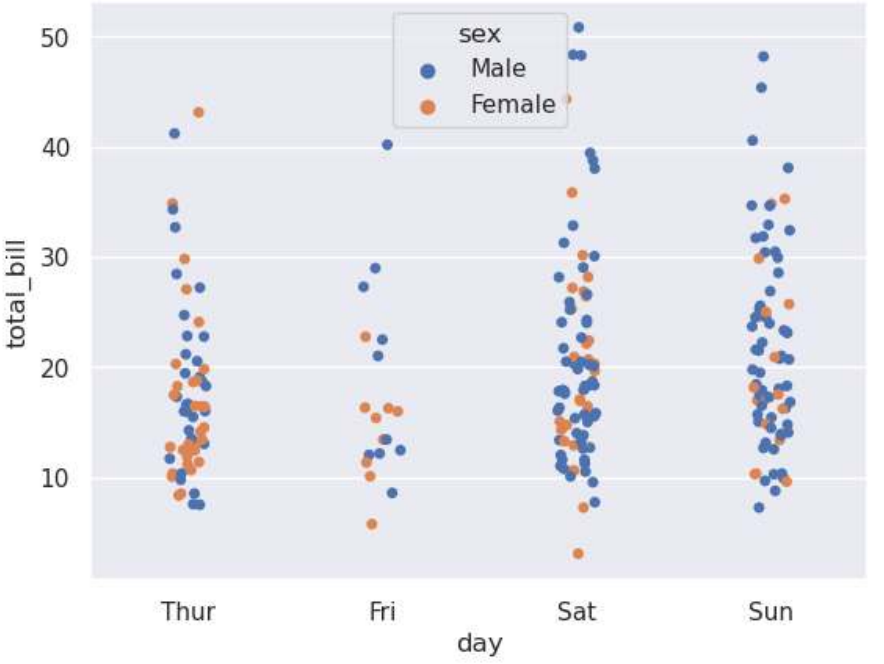

Figure 11: Image generated by the provided code.

Swarm Plot with `swarmplot()`

Swarm plots improve upon strip plots by automatically adjusting point positions to avoid overlaps.

Basic swarm plot

```
1  sns.swarmplot(data=tips, x="day", y="total_bill")
2  plt.show()
```

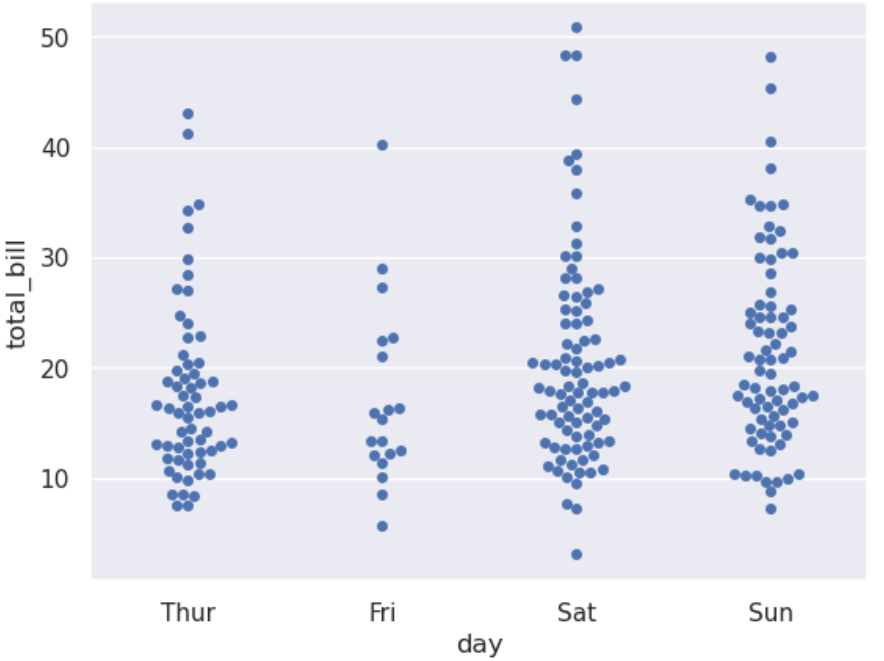

Figure 12: Image generated by the provided code.

Explanation:

Each point represents an observation, carefully placed to avoid collisions.

Swarm plot with hue

```
sns.swarmplot(data=tips, x="day", y="total_bill", hue="
    sex")
plt.show()
```

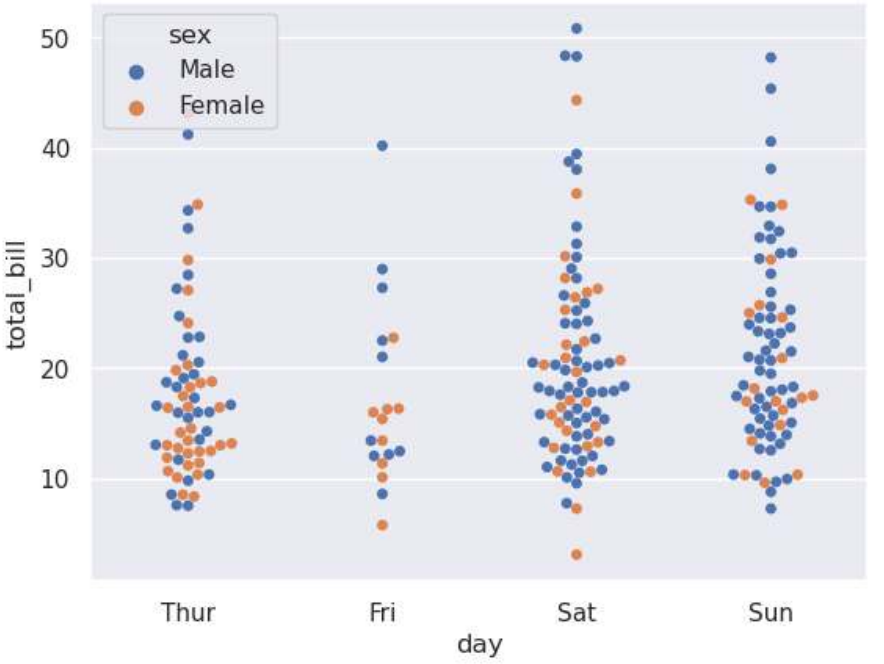

Figure 13: Image generated by the provided code.

Choosing the Right Plot

Plot	When to use
`barplot()`	When you want to show **aggregated statistics** (mean, median) by category
`countplot()`	When you want to show **counts** of observations per category
`boxplot()`	When you want to show **distribution summary** and detect **outliers**
`violinplot()`	When you want to show **distribution shape** and **summary statistics** together
`stripplot()`	When you want to show **all data points** for small datasets
`swarmplot()`	Same as `stripplot()` but better handling of **overlapping points**

Notes and Tips

- Combining plots is common: `boxplot()` + `stripplot()` or `violinplot()` + `swarmplot()` create more informative plots.
- Be mindful of readability when plotting many categories.
- Always check if `hue` improves or clutters your visualization.

In the next chapter, we will explore **Regression Plots**, which are essential when you want to model and visualize linear or non-linear relationships between variables.

Regression Plots

Introduction

Regression plots are designed to visualize the relationship between two variables, often to reveal and communicate trends or patterns. They are commonly used to:

- Visualize linear relationships.
- Detect non-linear patterns.
- Communicate the strength of a relationship.
- Show model-based predictions.

Seaborn provides two main functions for regression plots:

- `regplot()`: Low-level function that creates a scatter plot with a regression line.
- `lmplot()`: High-level function with additional options for faceting and easy grouping.

Simple Regression with `regplot()`

Basic regression plot

```
sns.regplot(data=tips, x="total_bill", y="tip")
plt.show()
```

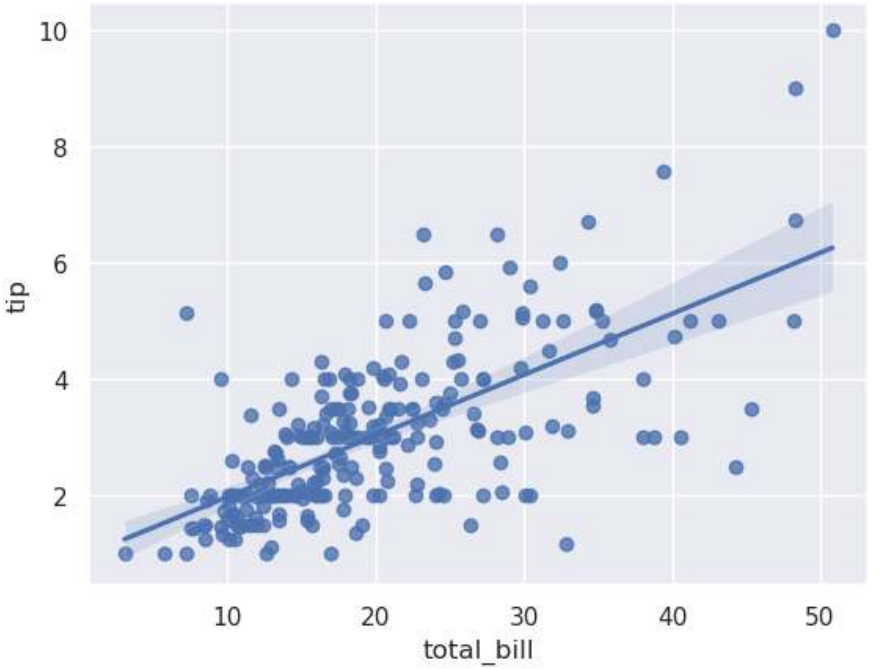

Figure 1: Image generated by the provided code.

Explanation:

This plot shows a scatter plot of `total_bill` vs `tip` with a fitted linear regression line and a 95% confidence interval by default.

Removing the confidence interval (`ci=None`)

```
sns.regplot(data=tips, x="total_bill", y="tip", ci=None)
plt.show()
```

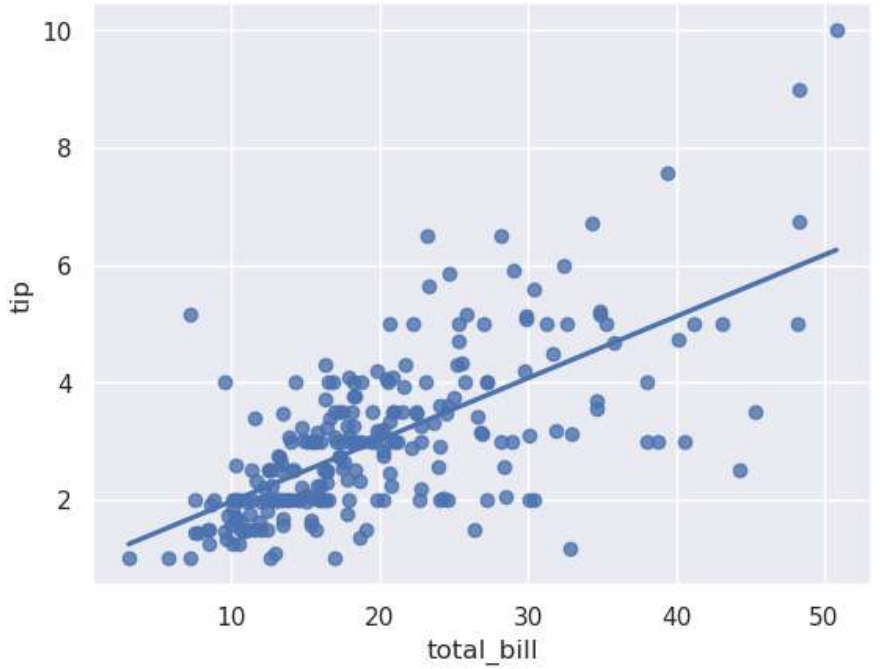

Figure 2: Image generated by the provided code.

This simplifies the plot by removing the shaded confidence interval.

Changing the order of the regression

By default, Seaborn fits a linear model, but you can fit higher-order polynomials.

```
sns.regplot(data=tips, x="total_bill", y="tip", order=2)
```

```
plt.show()
```

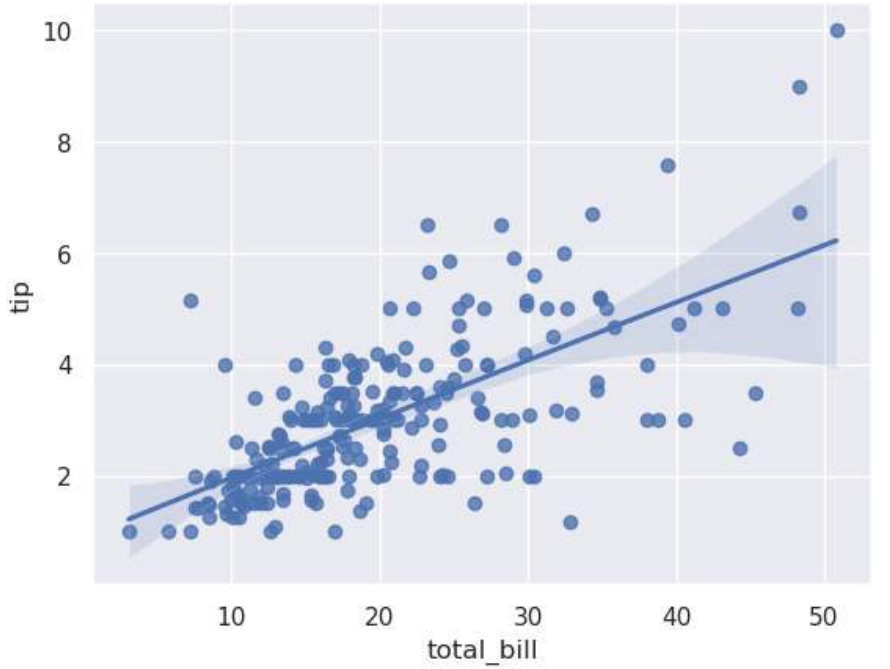

Figure 3: Image generated by the provided code.

Explanation:

The `order=2` argument fits a quadratic regression line.

Robust regression

You can make the regression robust to outliers by using `robust=True`.

```
1 sns.regplot(data=tips, x="total_bill", y="tip", robust=
      True)
2 plt.show()
```

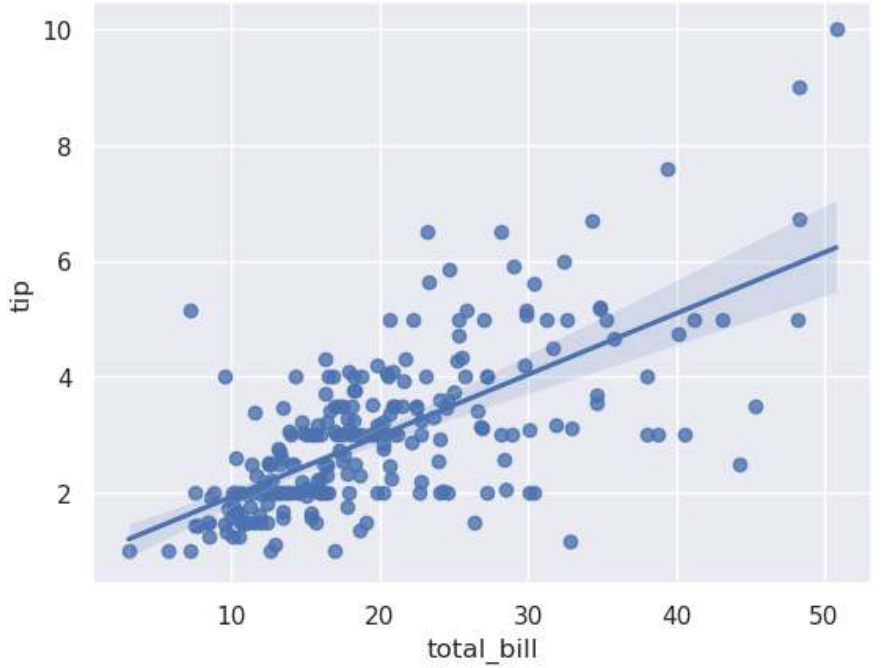

Figure 4: Image generated by the provided code.

Grouped Regression with `lmplot()`

Unlike `regplot()`, `lmplot()` allows easy grouping and faceting.

Basic `lmplot()`

```
1  sns.lmplot(data=tips, x="total_bill", y="tip")
2  plt.show()
```

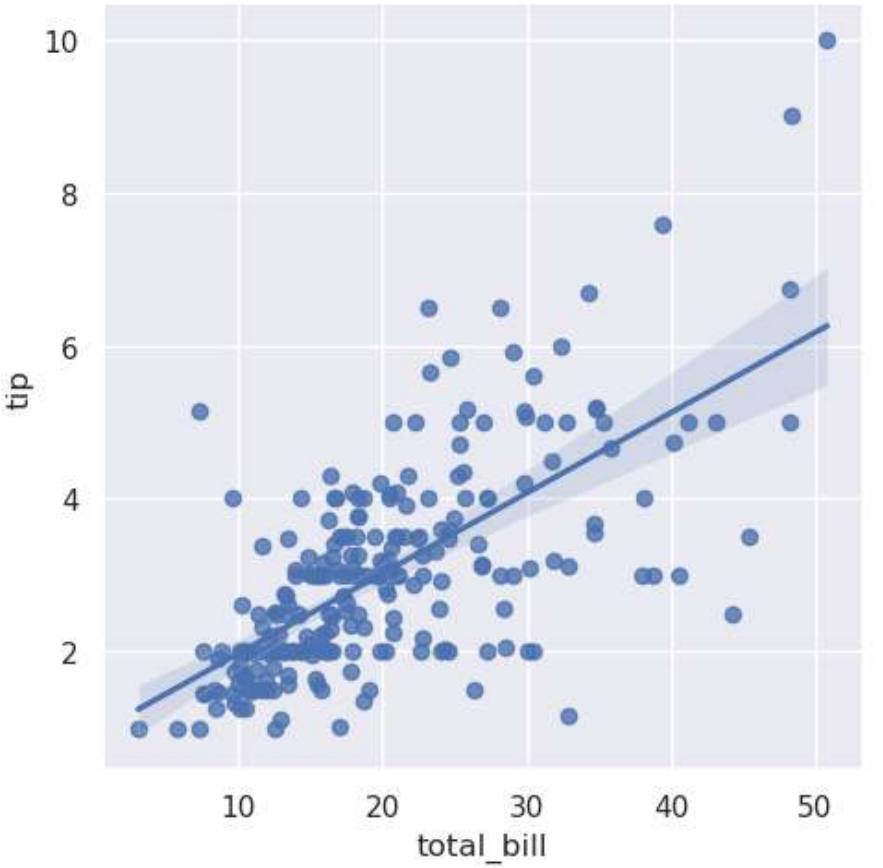

Figure 5: Image generated by the provided code.

Produces the same result as `regplot()` but returns a FacetGrid object.

Regression with hue

```
sns.lmplot(data=tips, x="total_bill", y="tip", hue="sex")
plt.show()
```

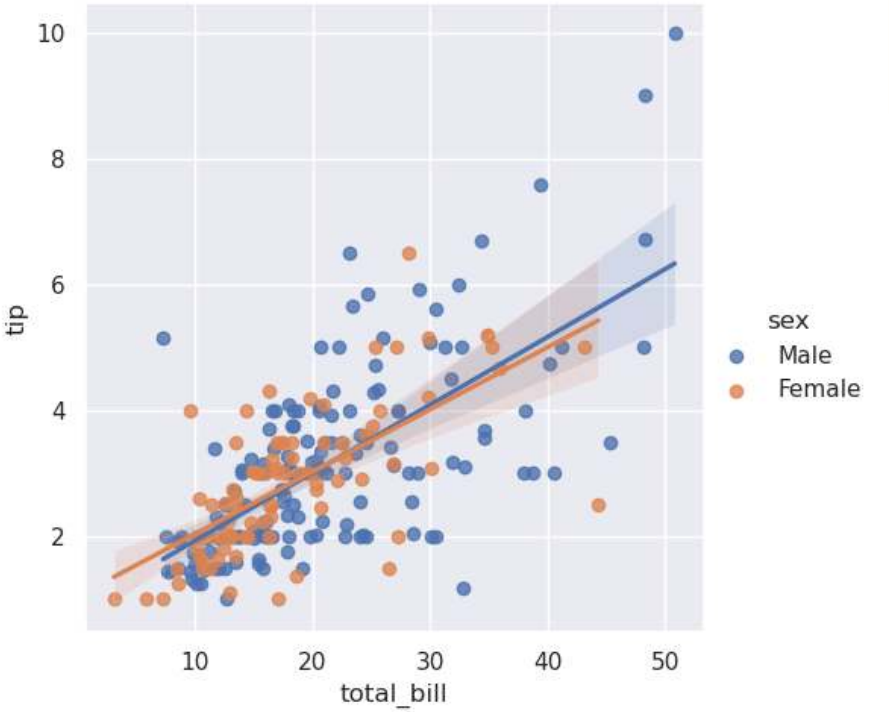

Figure 6: Image generated by the provided code.

Explanation:

Each group (`sex`) has its own regression line and scatter points.

Changing markers

```
sns.lmplot(data=tips,
           x="total_bill",
           y="tip",
           hue="sex",
           markers=["o", "s"])
plt.show()
```

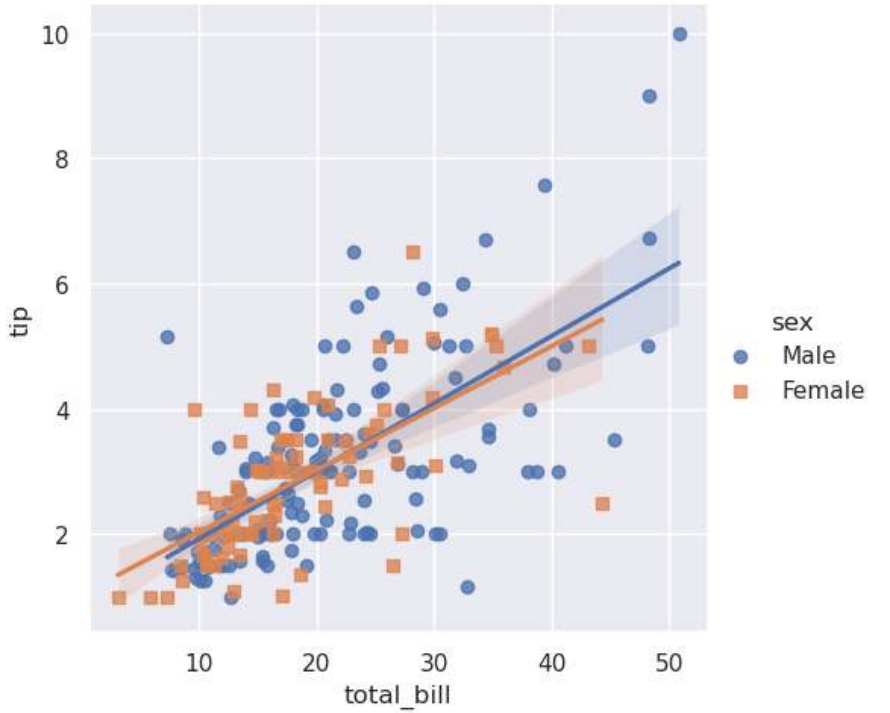

Figure 7: Image generated by the provided code.

You can assign different markers to each group for better readability.

Faceting with `col` and `row`

```
sns.lmplot(data=tips,
           x="total_bill",
           y="tip",
           col="sex",
           row="smoker")
plt.show()
```

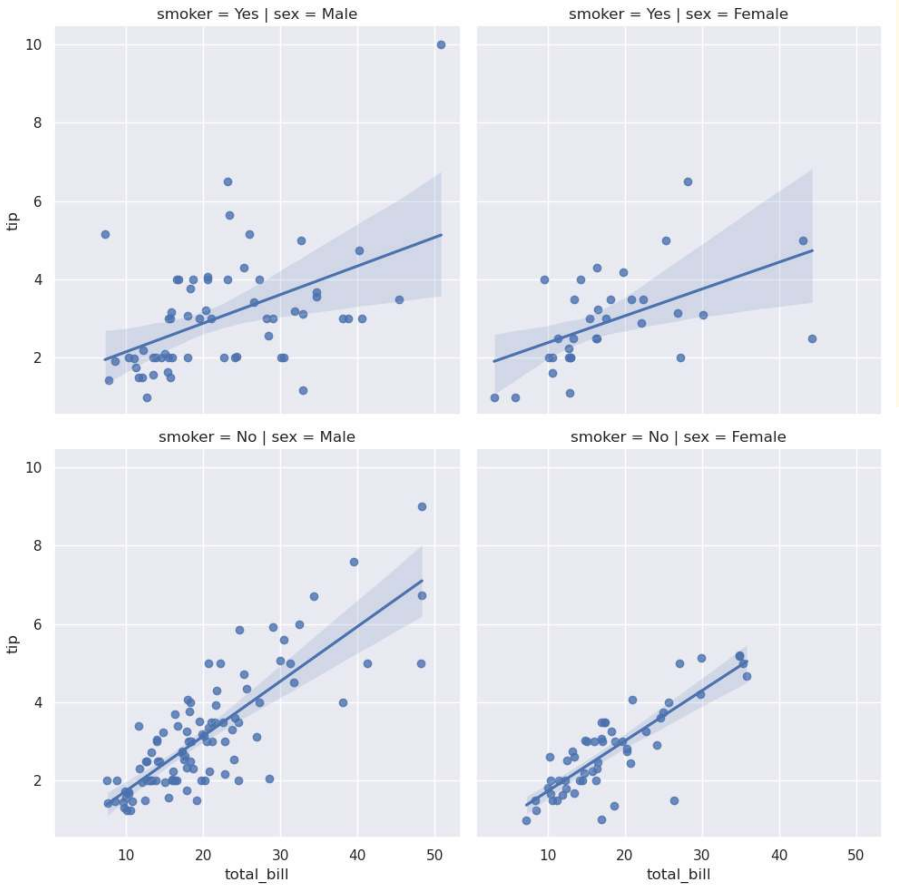

Figure 8: Image generated by the provided code.

Explanation:

This is a powerful feature: you can split the data into multiple panels by categories.

Multiple Regression (regression with additional predictors)

Seaborn does not directly support multiple regression in the statistical sense (with multiple `x` variables), but you can still:

1. Split data using `hue` to simulate group-wise multiple regression.
2. Use color, size, or style to show additional information.

Example: Multiple regression visualization

```
sns.lmplot(
    data=tips,
    x="total_bill",
    y="tip",
    hue="sex",
    col="smoker",
    scatter_kws={"s": 50},
    line_kws={"linewidth": 2}
)
plt.show()
```

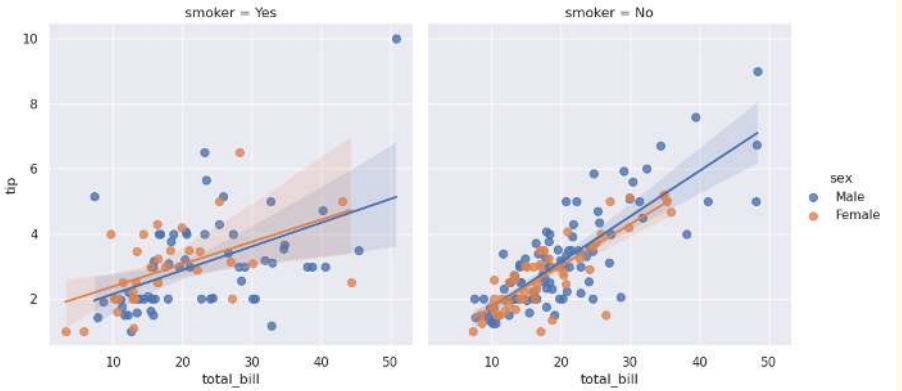

Figure 9: Image generated by the provided code.

Explanation:

This plot compares regression lines by both `sex` and `smoker` status, giving insight into the combined effect.

Arguments Summary

Argument	Description
`order`	Polynomial regression order
`ci`	Confidence interval (default 95%)
`hue`	Group by a categorical variable
`robust`	Robust regression to reduce outlier influence
`markers`	Specify marker style(s)

Argument	Description
`col` / `row`	Faceting by categories (lmplot only)
`scatter_kws`	Customize scatterplot aesthetics
`line_kws`	Customize regression line aesthetics

When to use `regplot()` vs. `lmplot()`

Function	Recommended for
`regplot()`	Simple regression plots without grouping or faceting
`lmplot()`	Regression plots involving grouping (`hue`) or faceting (`col`, `row`)

Notes

- Always check if a linear model is appropriate. In many real-world cases, relationships may be non-linear.
- Robust regression is helpful when you suspect outliers are affecting the model.
- Avoid overfitting when increasing `order` for polynomial regressions.

In the next chapter, we will explore **Matrix and Heatmap Plots**, which are ideal for visualizing relationships between many variables, especially correlations.

Matrix and Heatmap Plots

Introduction

Matrix plots are essential for visualizing **structured data** such as:

- Correlation matrices
- Distance matrices
- Contingency tables
- Any two-dimensional array of values

Seaborn offers two main functions for matrix visualizations:

- `heatmap()`: Displays a matrix with colored cells, optionally annotated.
- `clustermap()`: Extends `heatmap()` by applying clustering (hierarchical) to rows and/or columns automatically.

These plots are commonly used to identify patterns, clusters, or strong relationships between variables.

The Correlation Matrix

A typical use of matrix plots is to visualize the **correlation** between numerical variables.

Let's use the `penguins` dataset:

```
# Drop rows with missing values to compute correlation
penguins_clean = penguins.dropna()

# Compute correlation matrix
corr = penguins_clean.corr(numeric_only=True)
corr
```

	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g
bill_length_mm	1	-0.228626	0.653096	0.589451
bill_depth_mm	-0.228626	1	-0.577792	-0.472016
flipper_length_mm	0.653096	-0.577792	1	0.872979
body_mass_g	0.589451	-0.472016	0.872979	1

Basic Heatmap with `heatmap()`

Simple heatmap

```
sns.heatmap(corr)
plt.show()
```

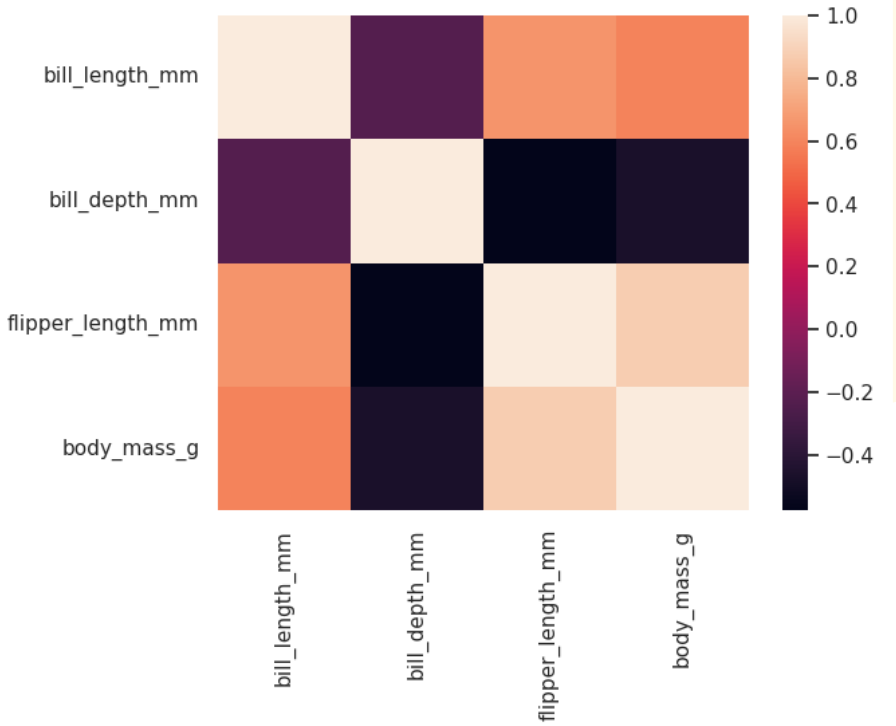

Figure 1: Image generated by the provided code.

Explanation:

- Darker colors represent stronger correlations.
- By default, Seaborn uses a blue color palette.

Adding annotations

```
1  sns.heatmap(corr, annot=True)
```

```
2  plt.show()
```

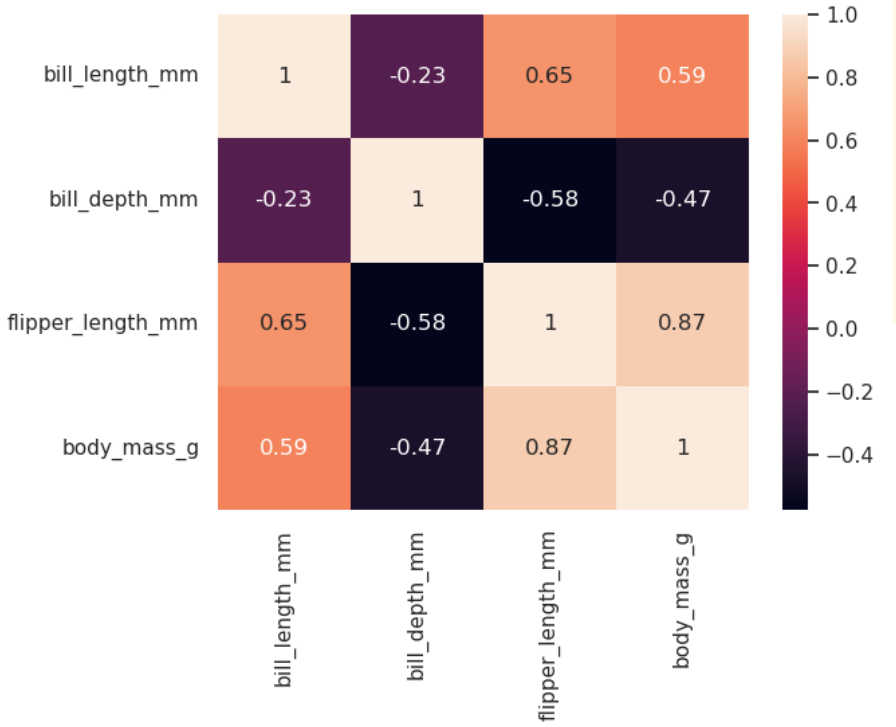

Figure 2: Image generated by the provided code.

Explanation:

`annot=True` displays the correlation coefficient inside each cell.

Customizing the color map

```
sns.heatmap(corr, annot=True, cmap="coolwarm", center=0)
plt.show()
```

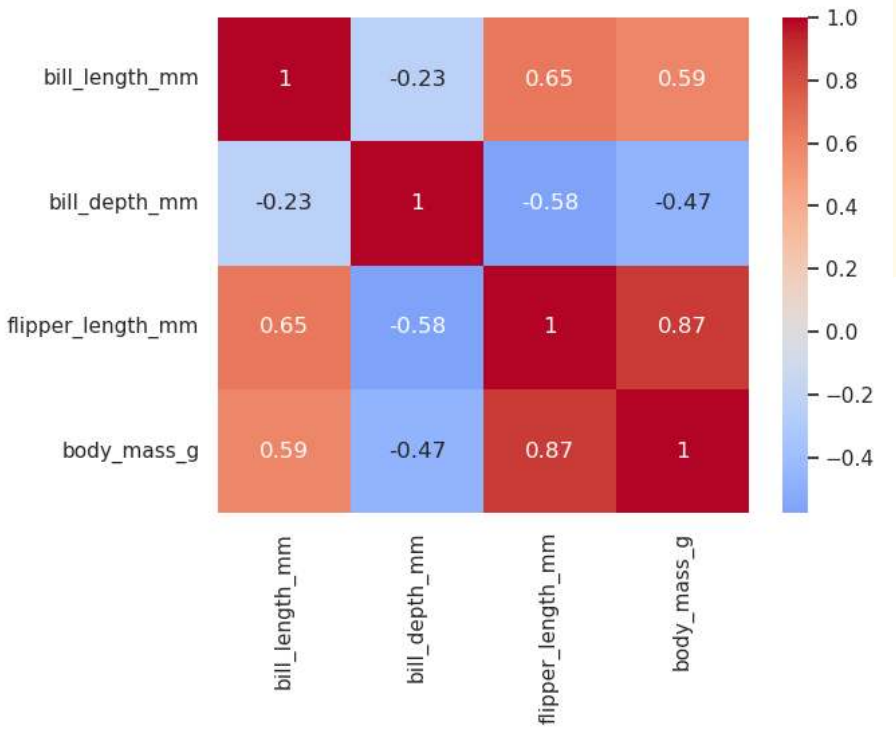

Figure 3: Image generated by the provided code.

Explanation:

- `cmap="coolwarm"` helps highlight positive and negative correlations.
- `center=0` ensures that the color map is symmetric around zero.

Adjusting aesthetics

```
sns.heatmap(corr,
            annot=True,
            fmt=".2f",
            linewidths=0.5,
            linecolor="gray",
            cbar_kws={"shrink": 0.8})
plt.show()
```

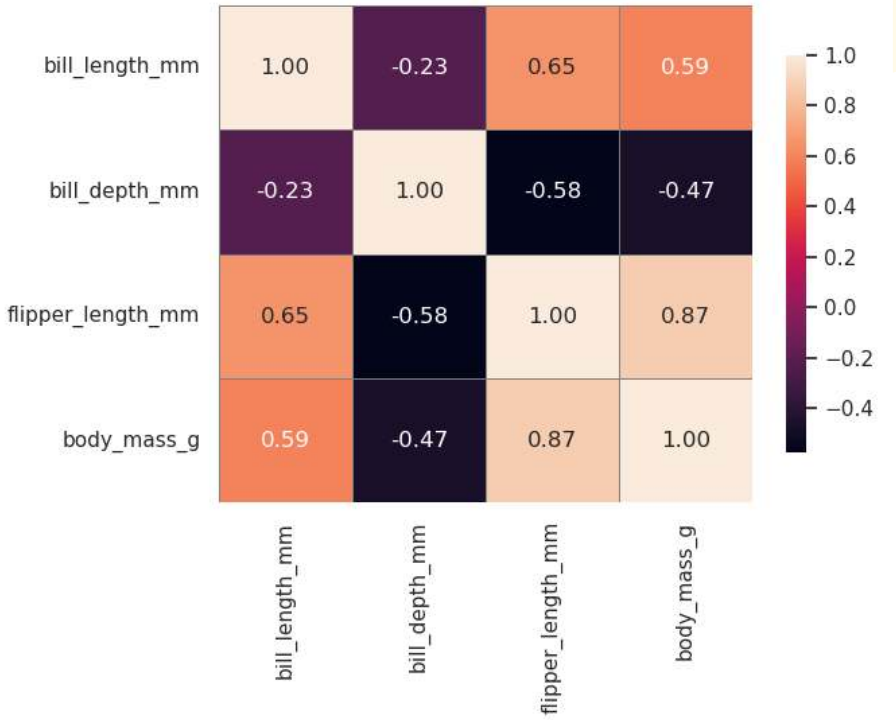

Figure 4: Image generated by the provided code.

Explanation:

- `fmt=".2f"` formats numbers with two decimals.
- `linewidths` and `linecolor` add grid lines between cells.
- `cbar_kws` adjusts the colorbar size.

Clustermap with `clustermap()`

`clustermap()` performs hierarchical clustering of rows and columns to reveal structures and groups in the data.

Basic clustermap

```
sns.clustermap(corr)
plt.show()
```

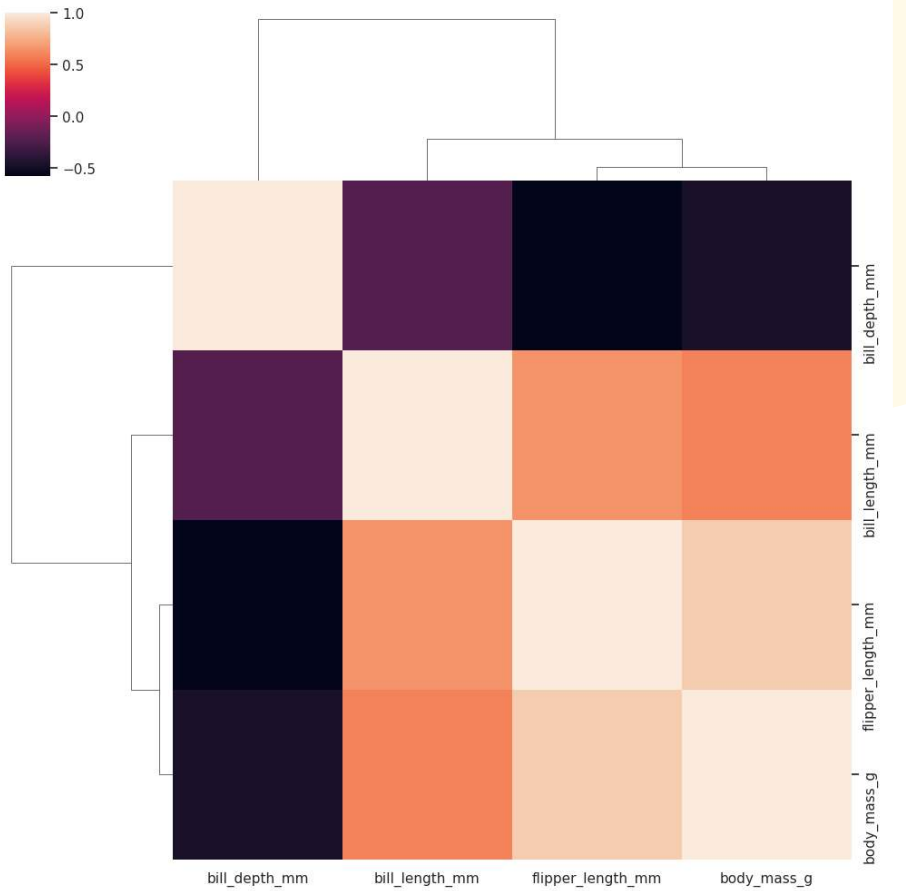

Figure 5: Image generated by the provided code.

Explanation:

- Variables are reordered to show clusters.
- Dendrograms are displayed to represent the hierarchical clustering.

Customizing clustermap

```
sns.clustermap(corr,
               cmap="coolwarm",
               annot=True,
               fmt=".2f",
               linewidths=0.5)
plt.show()
```

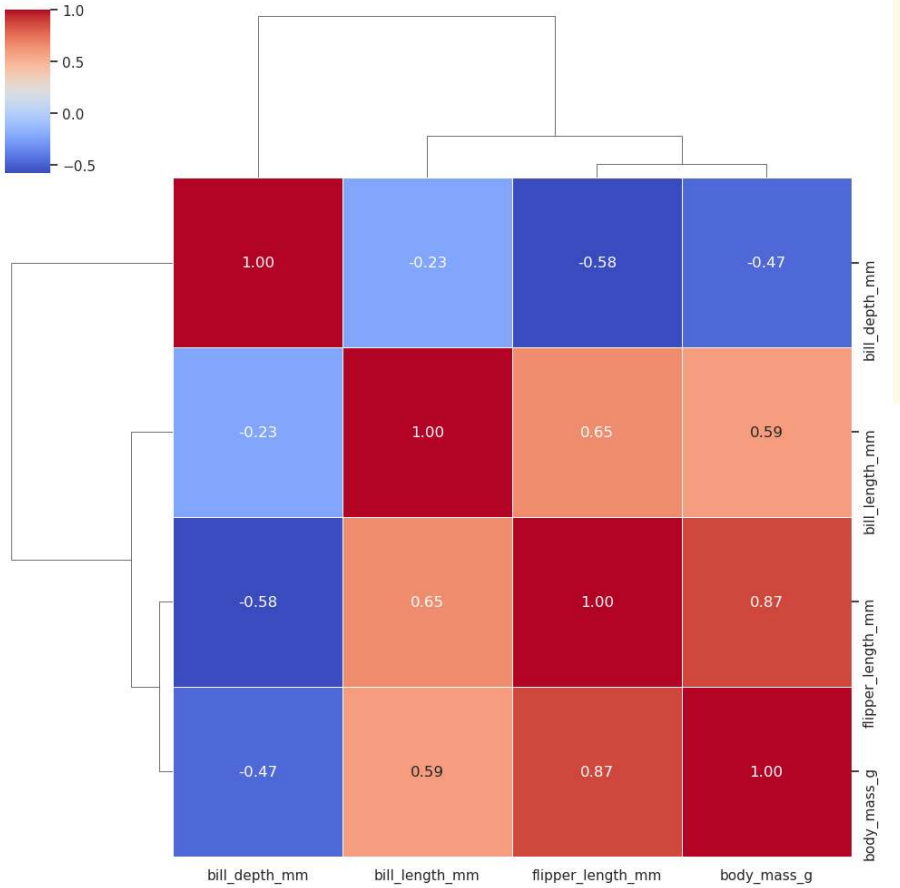

Figure 6: Image generated by the provided code.

Explanation:

You can pass most arguments from `heatmap()` to `clustermap()`.

Real-world Example: Flights dataset as a matrix

The `flights` dataset can be reshaped into a matrix using `pivot()`.

```
# Load dataset
flights = sns.load_dataset("flights")
flights_pivot = flights.pivot(index="month",
                              columns="year",
                              values="passengers")
flights_pivot
```

month	1949	1950	1951	1952	1953	1954	1955	1956	1957	1958	1959	1960
Jan	112	115	145	171	196	204	242	284	315	340	360	417
Feb	118	126	150	180	196	188	233	277	301	318	342	391
Mar	132	141	178	193	236	235	267	317	356	362	406	419
Apr	129	135	163	181	235	227	269	313	348	348	396	461
May	121	125	172	183	229	234	270	318	355	363	420	472
Jun	135	149	178	218	243	264	315	374	422	435	472	535
Jul	148	170	199	230	264	302	364	413	465	491	548	622
Aug	148	170	199	242	272	293	347	405	467	505	559	606
Sep	136	158	184	209	237	259	312	355	404	404	463	508
Oct	119	133	162	191	211	229	274	306	347	359	407	461
Nov	104	114	146	172	180	203	237	271	305	310	362	390
Dec	118	140	166	194	201	229	278	306	336	337	405	432

Heatmap of passengers

```python
sns.heatmap(flights_pivot,
            annot=True,
            fmt="d",
            cmap="coolwarm",
            linewidths=0.5,
            linecolor="white",
            annot_kws={"size":8})
plt.show()
```

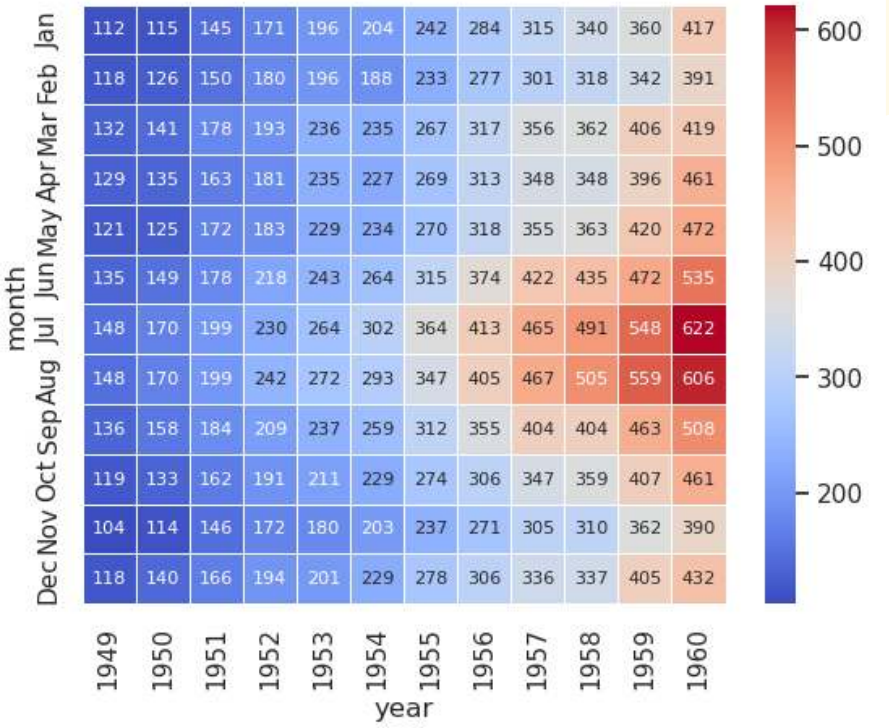

Figure 7: Image generated by the provided code.

Explanation:

- Rows are months, columns are years, and the values represent the number of passengers.
- This is a perfect example of a **time matrix**.

Clustermap on flights data

```
sns.clustermap(flights_pivot, cmap="coolwarm")
plt.show()
```

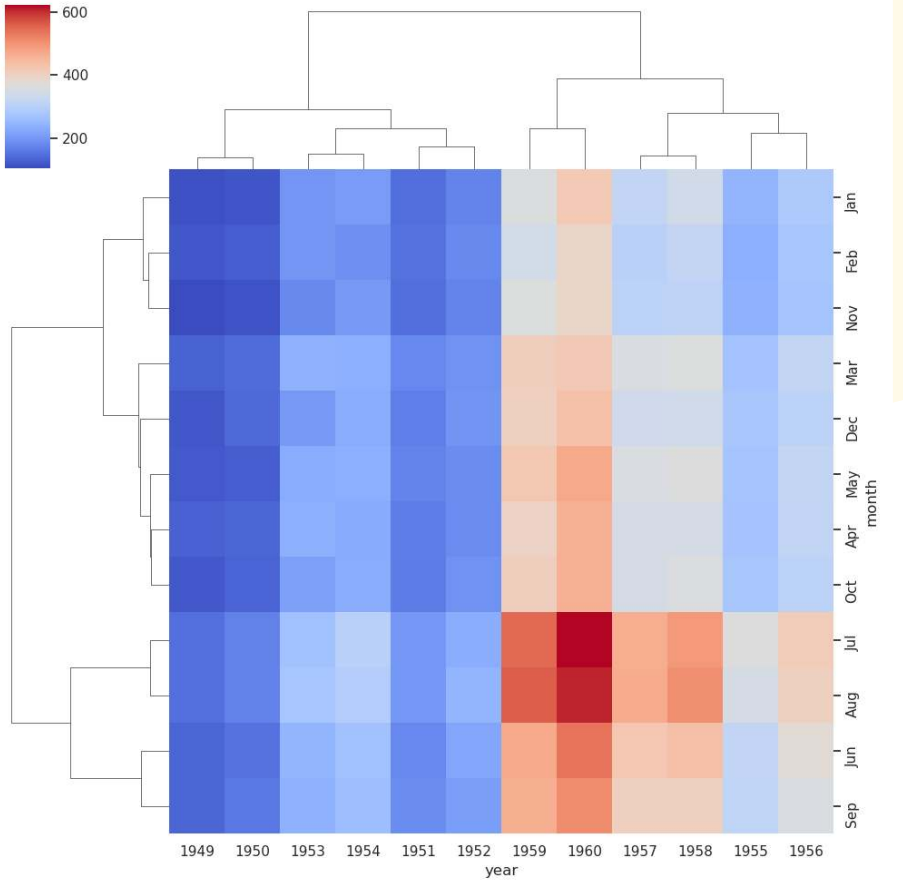

Figure 8: Image generated by the provided code.

Explanation:

This automatically clusters both months and years based on the number of passengers, revealing patterns such as similar years or seasonal clusters.

Notes and Tips

- `heatmap()` is great for **known matrices**, like correlations.
- `clustermap()` is helpful when you suspect the data might have **hidden structure**.
- Adjust the `cmap` to highlight the type of information you want to emphasize (diverging, sequential, etc.).
- Always label your axes clearly when reshaping data.

Summary Table

Function	Purpose
`heatmap()`	Plot any numeric matrix with customizable colors and annotations
`clustermap()`	Plot a matrix with hierarchical clustering applied automatically
`annot=True`	Display numerical values inside the cells
`cmap=`	Control the color palette
`fmt=`	Control the number formatting in annotations

In the next chapter, we will learn how to use **FacetGrid and PairGrid**, two essential tools for building complex grids of plots automatically.

Multiplot Grids

Introduction

One of Seaborn's strengths is its ability to easily create **multi-plot grids**, allowing you to:

- Explore distributions or relationships across subgroups.
- Create panel plots with shared or independent axes.
- Automate complex plotting layouts.

Seaborn provides three main tools for this:

- `FacetGrid()`: The most flexible way to create custom grids of plots.
- `pairplot()`: A quick way to visualize pairwise relationships between variables.
- `catplot()`: Combines categorical plots with faceting capabilities.

FacetGrid

Basic FacetGrid

```
g = sns.FacetGrid(tips, col="sex")
g.map(sns.histplot, "total_bill")
plt.show()
```

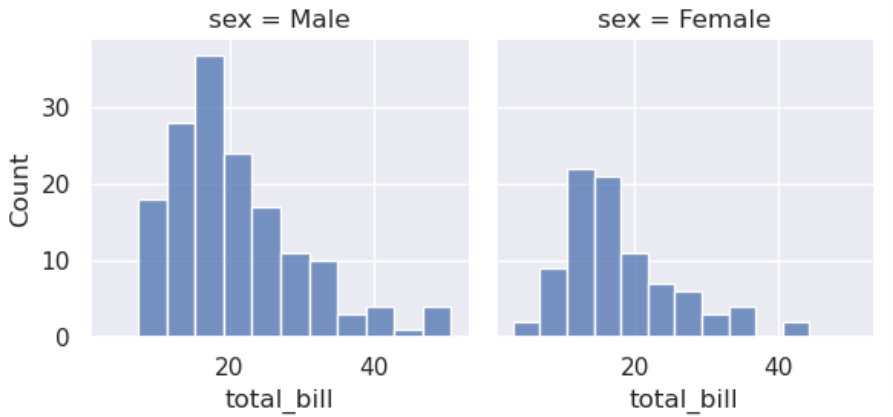

Figure 1: Image generated by the provided code.

Explanation:

- Creates one histogram for each value of `sex`.
- `map()` allows you to specify the plotting function to apply.

Faceting by row and column

```
g = sns.FacetGrid(tips, col="sex", row="smoker")
g.map(sns.scatterplot, "total_bill", "tip")
plt.show()
```

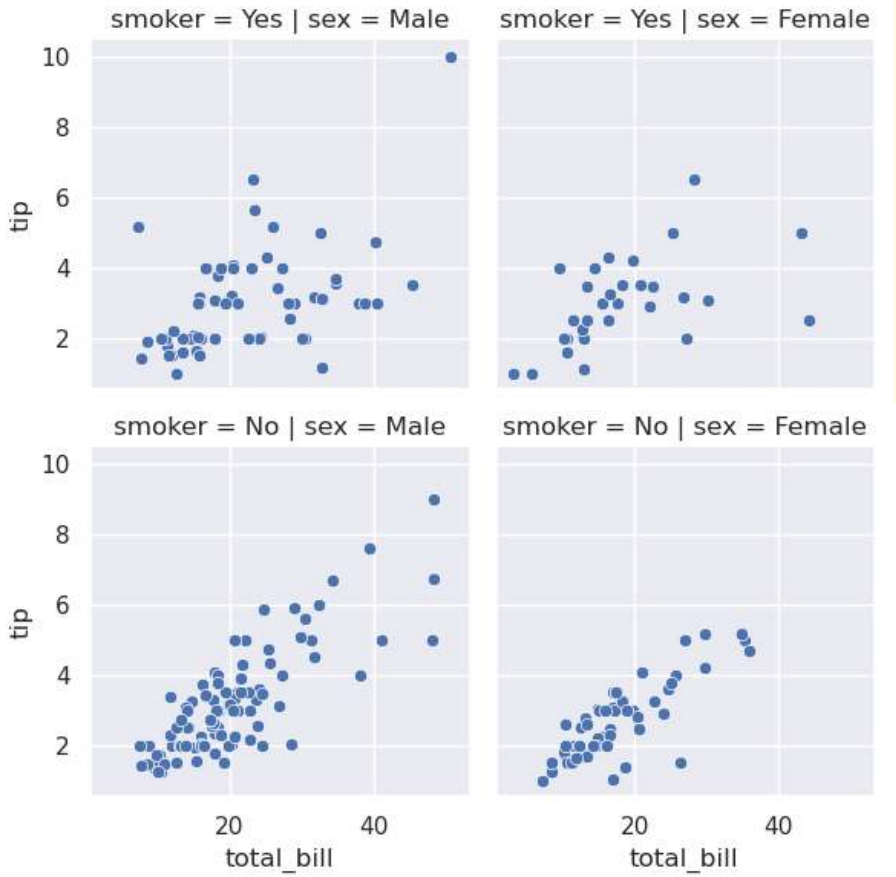

Figure 2: Image generated by the provided code.

Explanation:

- Creates a **grid of scatter plots** combining `sex` and `smoker`.
- Each cell of the grid corresponds to a subgroup.

Adding hue inside a FacetGrid

```
g = sns.FacetGrid(tips, col="day", hue="sex")
g.map(sns.scatterplot, "total_bill", "tip").add_legend()
plt.show()
```

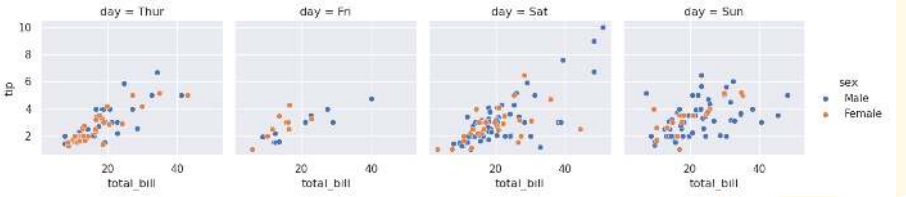

Figure 3: Image generated by the provided code.

Explanation:

- hue still works inside each facet, making comparisons easier.

Customizing plots inside FacetGrid

```
g = sns.FacetGrid(tips, col="day", height=4, aspect=0.7)
g.map_dataframe(sns.regplot,
                x="total_bill",
                y="tip",
                scatter_kws={"s": 30},
                line_kws={"color": "red"})
g.add_legend()
plt.show()
```

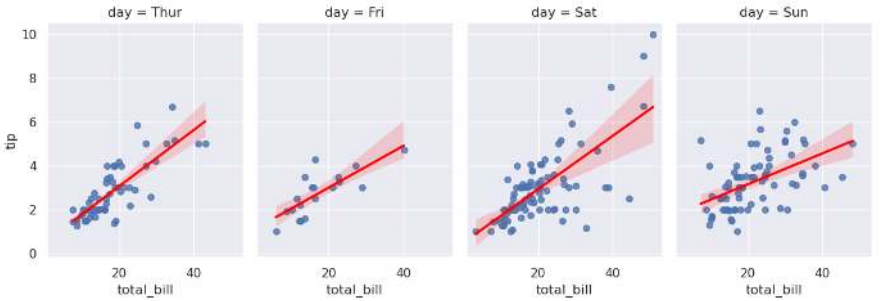

Figure 4: Image generated by the provided code.

Pairplot

The `pairplot()` function is a **quick and powerful** way to create scatterplot matrices.

Basic Pairplot

```
sns.pairplot(penguins.dropna())
plt.show()
```

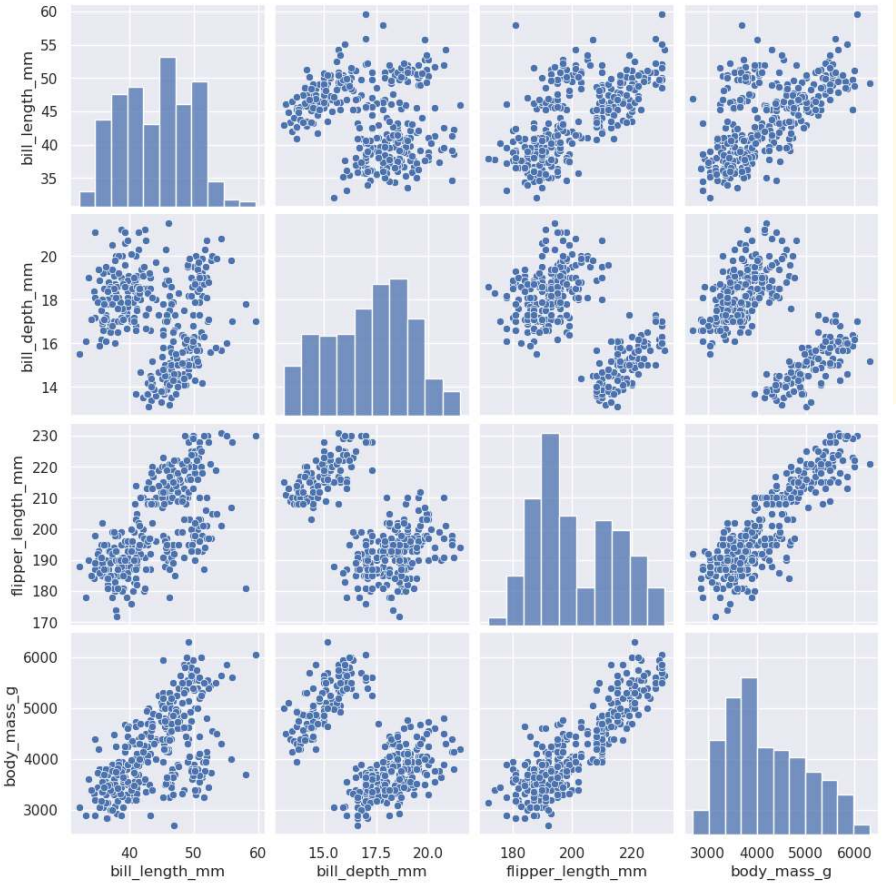

Figure 5: Image generated by the provided code.

Explanation:

- Automatically plots pairwise relationships between all numeric variables.
- Diagonal plots show the distribution of each variable.

Pairplot with hue

```
1  sns.pairplot(penguins.dropna(), hue="species")
2  plt.show()
```

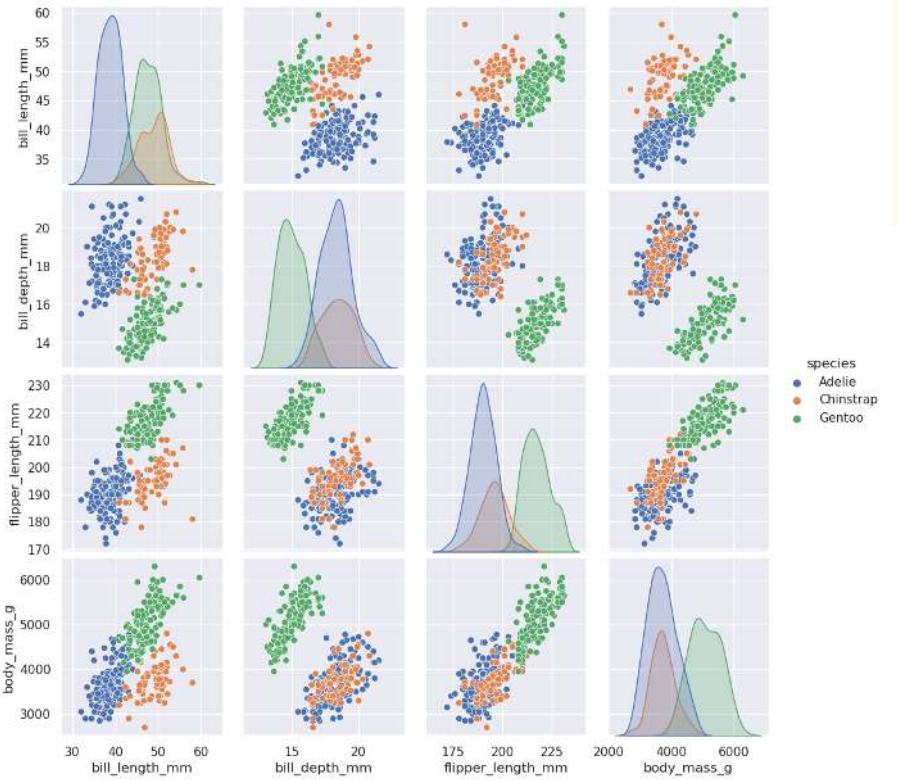

Figure 6: Image generated by the provided code.

Explanation:

- Colors the plots by species.

- Very helpful for detecting clusters and patterns.

Customizing pairplot markers

```
sns.pairplot(penguins.dropna(),
             hue="species",
             markers=["o", "s", "D"])
plt.show()
```

You can specify custom markers for each group.

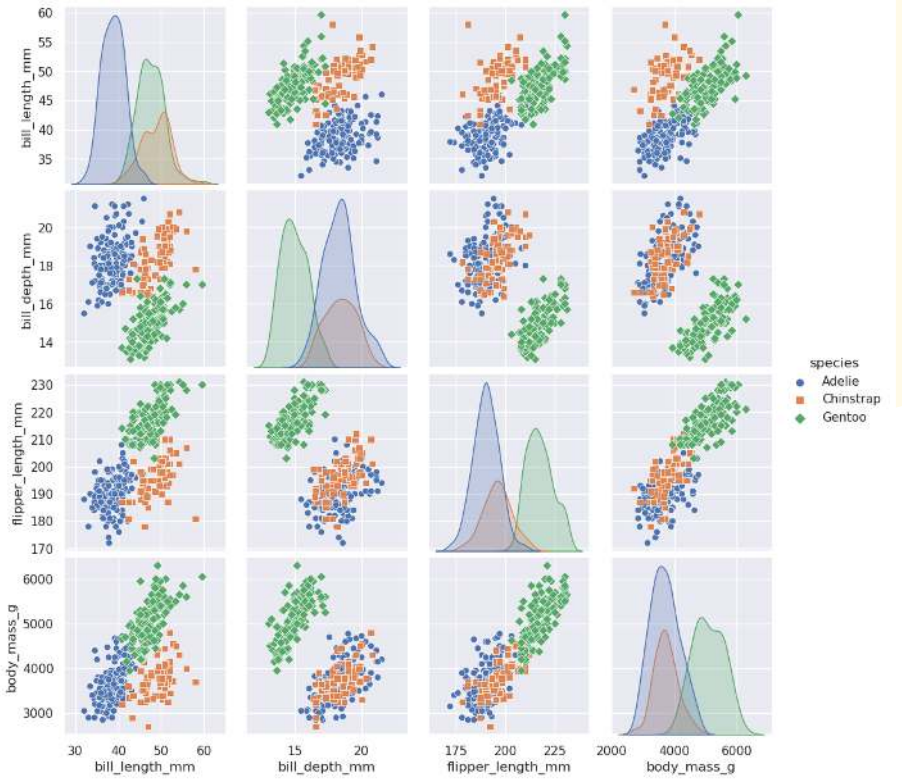

Figure 7: Image generated by the provided code.

Pairplot with regression lines

```
sns.pairplot(penguins.dropna(), hue="species", kind="reg"
    )
plt.show()
```

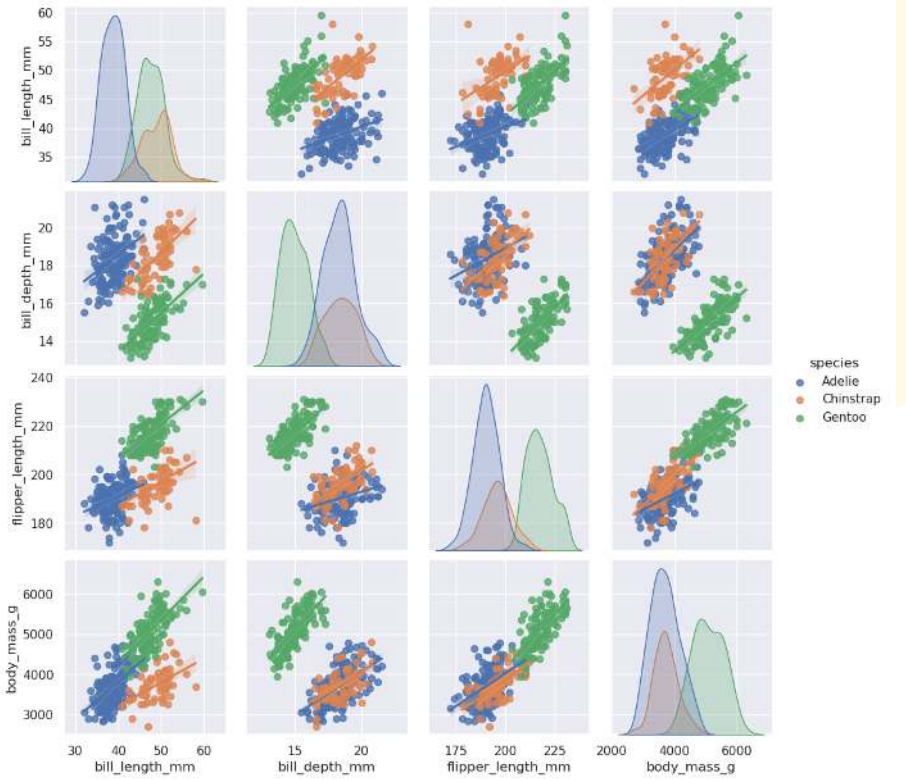

Figure 1: Image generated by the provided code.

Explanation:

- Adds regression lines to scatterplots instead of plain points.

Catplot

`catplot()` is a **general-purpose** function for categorical plots with faceting built-in.

Basic catplot (equivalent to barplot)

```
1 sns.catplot(data=tips, x="day", y="total_bill", kind="bar
      ")
2 plt.show()
```

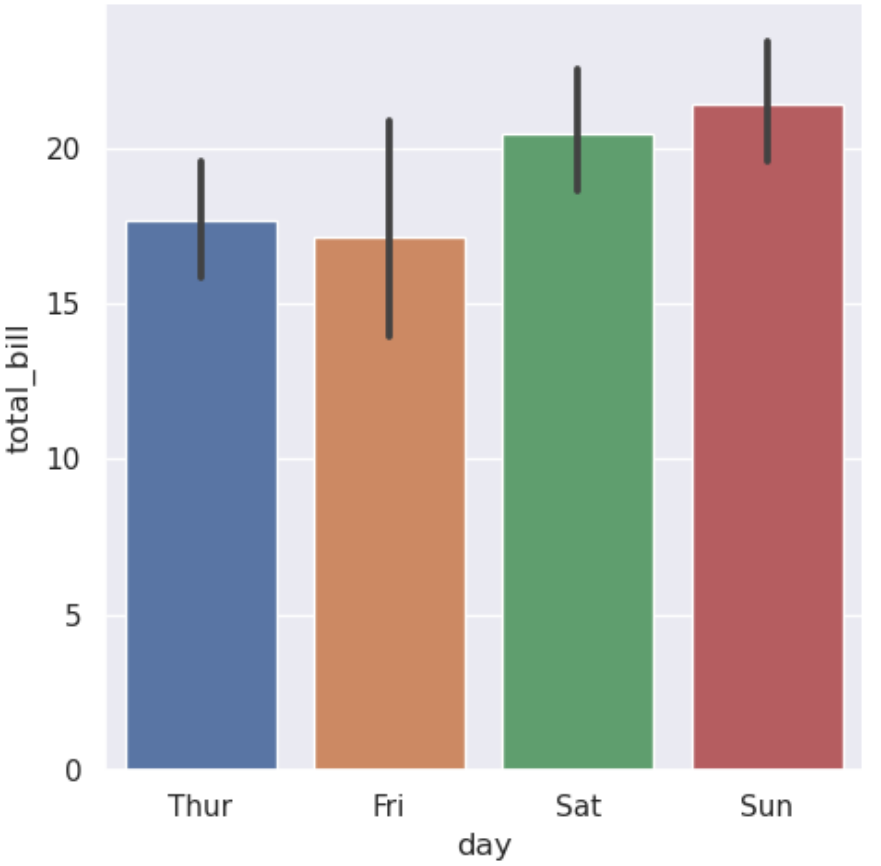

Figure 2: Image generated by the provided code.

Changing plot type

```
sns.catplot(data=tips, x="day", y="total_bill", kind="box")
plt.show()
```

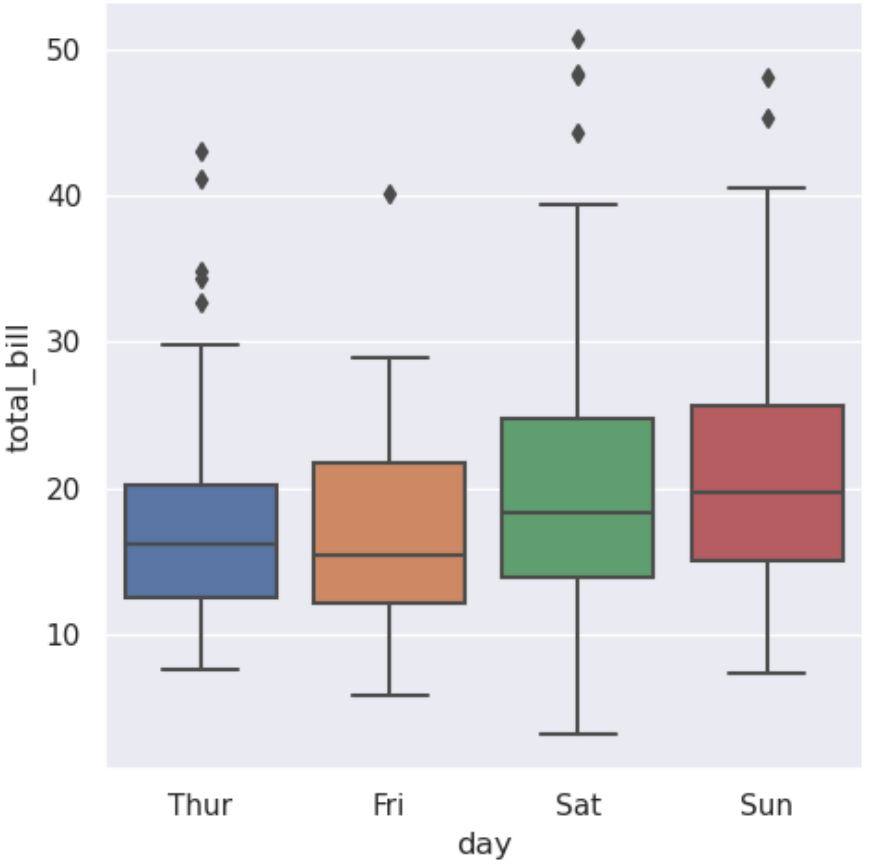

Figure 3: Image generated by the provided code.

You can switch between `box`, `violin`, `strip`, `swarm`, and `bar` just by changing the `kind`.

Faceting with catplot

```
sns.catplot(data=tips,
            x="day", y="total_bill",
            hue="sex",
            col="smoker", kind="box")
plt.show()
```

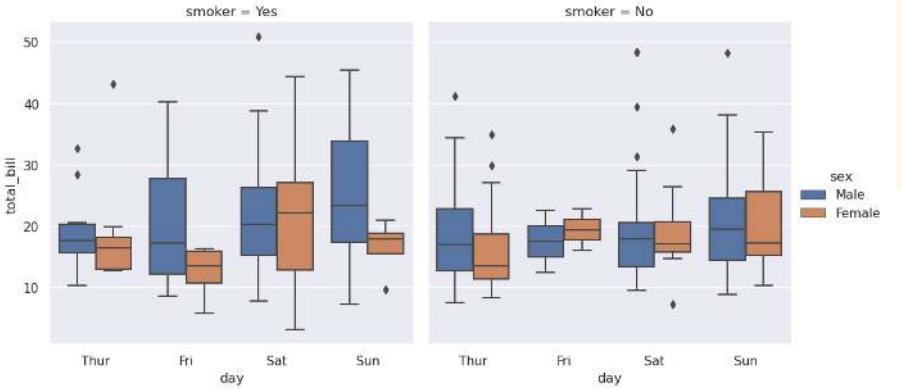

Figure 4: Image generated by the provided code.

Explanation:

- Creates a grid of boxplots split by `smoker`.
- Inside each plot, the `hue` separates the data by `sex`.

Changing orientation

```
sns.catplot(data=tips,
            y="day", x="total_bill",
            kind="violin", hue="sex", split=True)
plt.show()
```

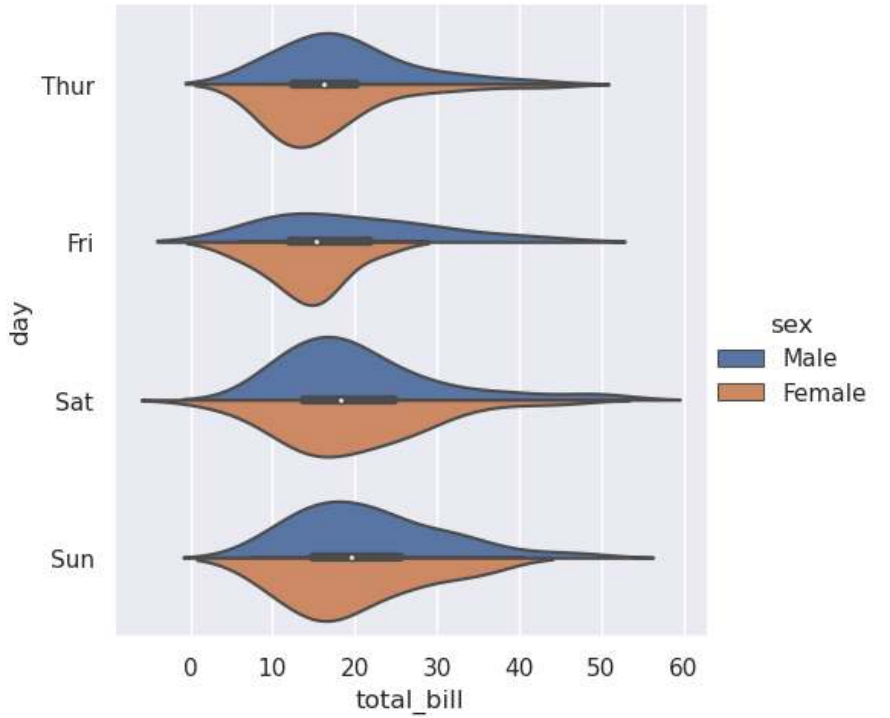

Figure 5: Image generated by the provided code.

By switching `x` and `y`, you change the orientation of the plot.

Notes and Tips

- `FacetGrid()` gives you full control but requires more manual setup.
- `pairplot()` is excellent for exploratory data analysis with numerical variables.

- `catplot()` is very flexible for categorical data and is often the easiest choice for fast, publication-ready plots.
- Use `height` and `aspect` to adjust the size of facets.
- Always check if too many facets make the plot harder to interpret.

Summary Table

Function	Use case
`FacetGrid()`	Full control for creating custom grid plots
`pairplot()`	Automatic pairwise scatterplots for numerical variables
`catplot()`	Categorical plots + automatic faceting

In the next chapter, we will focus on **Advanced Customization**, where you will learn how to change themes, palettes, and styles to make your plots publication-quality.

Advanced Customization

Introduction

One of the greatest strengths of Seaborn is that it not only makes it easy to generate statistical plots but also provides tools to make them **visually attractive and publication-ready**.

In this chapter, you will learn how to: - Change plot themes and styles. - Customize color palettes. - Adjust contexts to match different audiences (notebooks, papers, presentations). - Combine all these options for fully customized visualizations.

Changing Themes with `set_theme()`

Seaborn comes with several built-in themes you can use to quickly adjust the overall appearance of your plots.

Available themes:

- `"darkgrid"` (default)
- `"whitegrid"`
- `"dark"`
- `"white"`
- `"ticks"`

Example: Switching themes

```
sns.set_theme(style="whitegrid")
sns.boxplot(data=tips, x="day", y="total_bill")
plt.show()
```

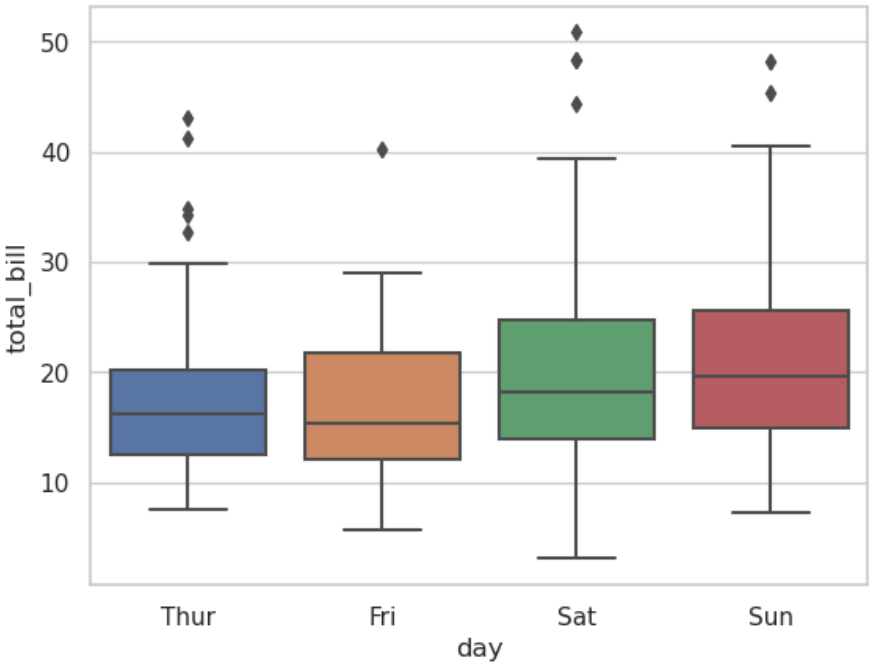

Figure 1: Image generated by the provided code.

Explanation:

`whitegrid` is commonly used in scientific publications as it provides clear gridlines.

Manually Changing Style Elements

If you want more control, you can adjust specific style elements.

Example:

```
sns.set_style("whitegrid")
sns.set_context("talk", font_scale=1.2)
sns.boxplot(data=tips, x="day", y="total_bill")
plt.show()
```

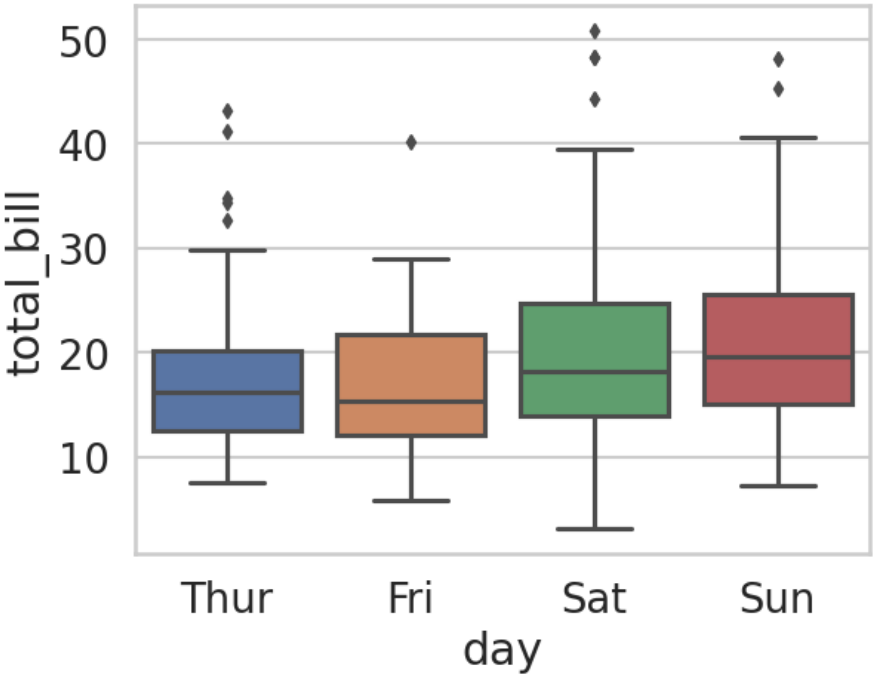

Figure 2: Image generated by the provided code.

- `set_context("talk")` adapts the plot for presentations.
- `font_scale` allows you to adjust the size of text elements globally.

Color Palettes with `set_palette()`

Colors play a crucial role in data visualization. Seaborn provides predefined palettes and also allows you to create custom ones.

Example: Default palette

```
sns.set_theme()
sns.boxplot(data=tips, x="day", y="total_bill", hue="sex"
    )
plt.show()
```

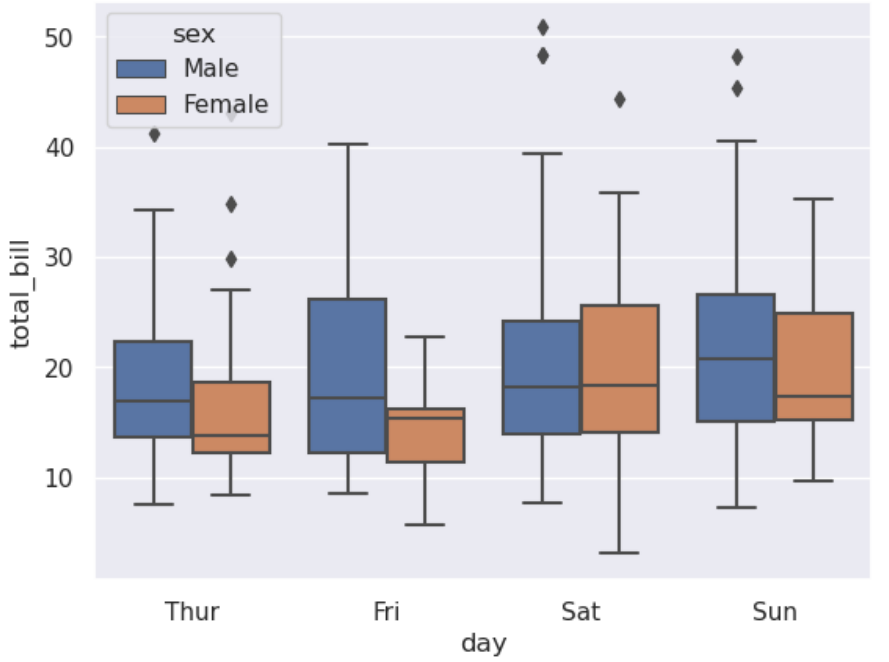

Figure 3: Image generated by the provided code.

Changing the palette

```
1 sns.set_palette("pastel")
2 sns.boxplot(data=tips, x="day", y="total_bill", hue="sex"
      )
3 plt.show()
```

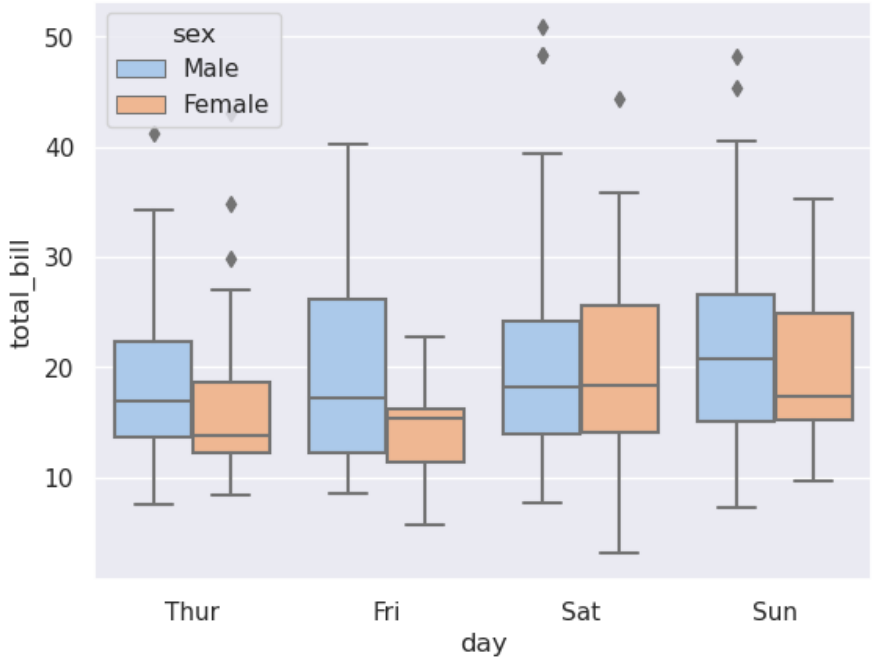

Figure 4: Image generated by the provided code.

Example: Using a sequential palette

```
sns.set_palette("Blues")
sns.violinplot(data=tips,
               x="day", y="total_bill",
               hue="sex", split=True)
plt.show()
```

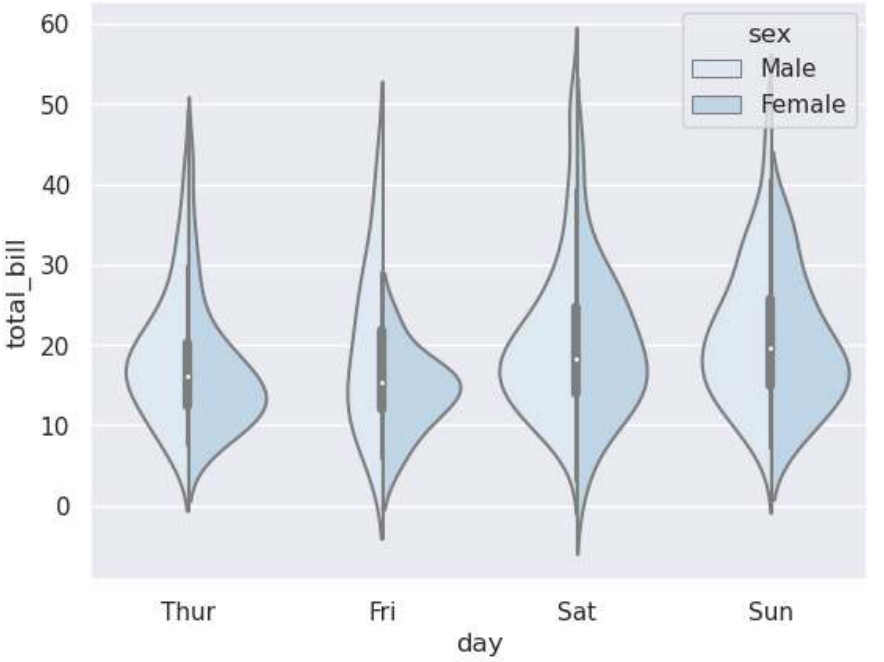

Figure 5: Image generated by the provided code.

Example: Custom palette

```
custom_palette = ["#FF6F61", "#6B5B95"]
sns.set_palette(custom_palette)
sns.barplot(data=tips, x="day", y="total_bill", hue="sex"
    )
plt.show()
```

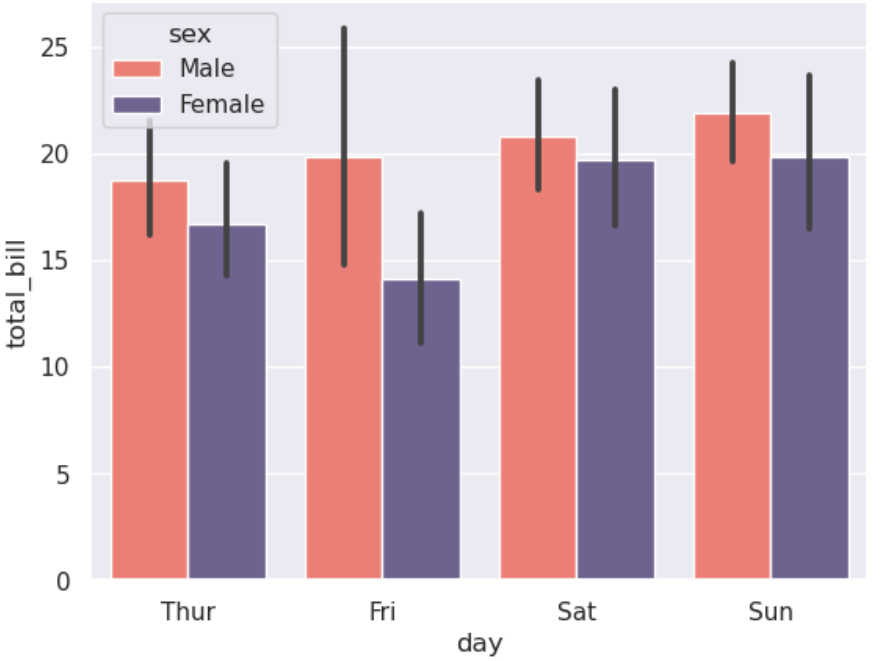

Figure 6: Image generated by the provided code.

Contexts for Different Purposes

Seaborn allows you to adjust the **scale** of elements depending on your target audience.

Available contexts:

- `"paper"` (smaller plots for papers)
- `"notebook"` (default)

- `"talk"` (suitable for presentations)
- `"poster"` (for large visuals)

Example: Switching contexts

```
sns.set_context("poster")
sns.barplot(data=tips, x="day", y="total_bill", hue="sex")
plt.show()
```

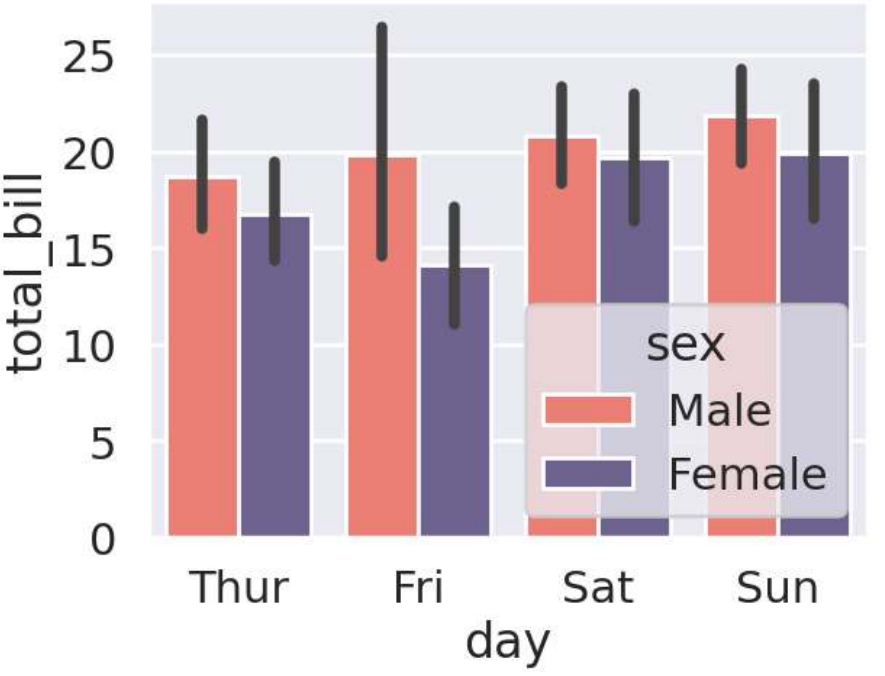

Figure 7: Image generated by the provided code.

Explanation:

Notice how all elements (text, lines, markers) are larger, suitable for projecting slides or posters.

Combining style, palette, and context

```
sns.set_theme(style="ticks",
              palette="Set2",
              context="talk")
sns.violinplot(data=tips,
               x="day", y="total_bill", hue="sex",
               split=True)
plt.show()
```

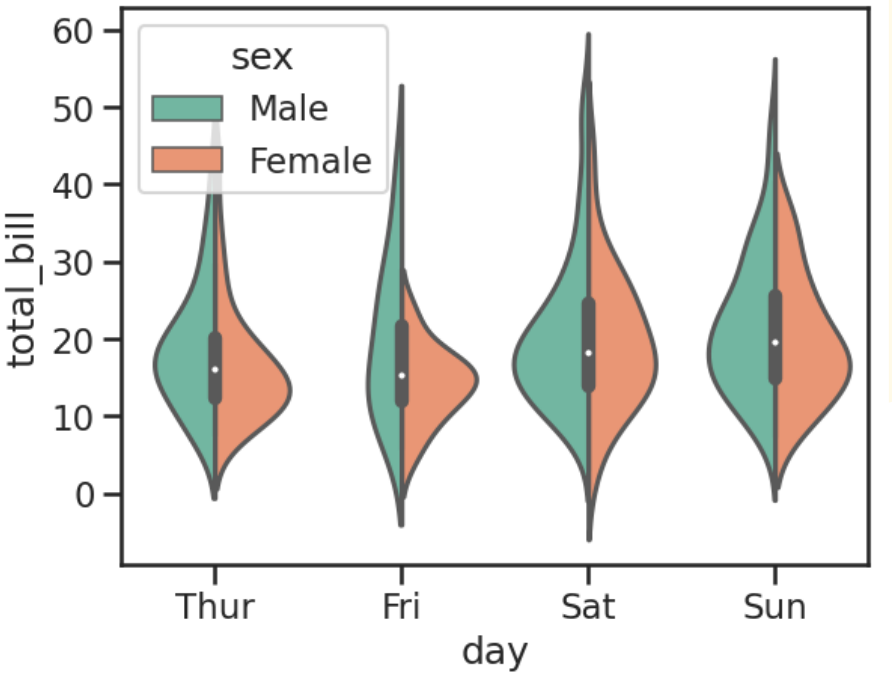

Figure 8: Image generated by the provided code.

By combining style, palette, and context, you can create customized plots adapted to your specific needs.

Temporary Customization using `with`

You can apply temporary changes to a specific plot without affecting others:

```
with sns.axes_style("whitegrid"):
    sns.boxplot(data=tips, x="day", y="total_bill")
    plt.show()
```

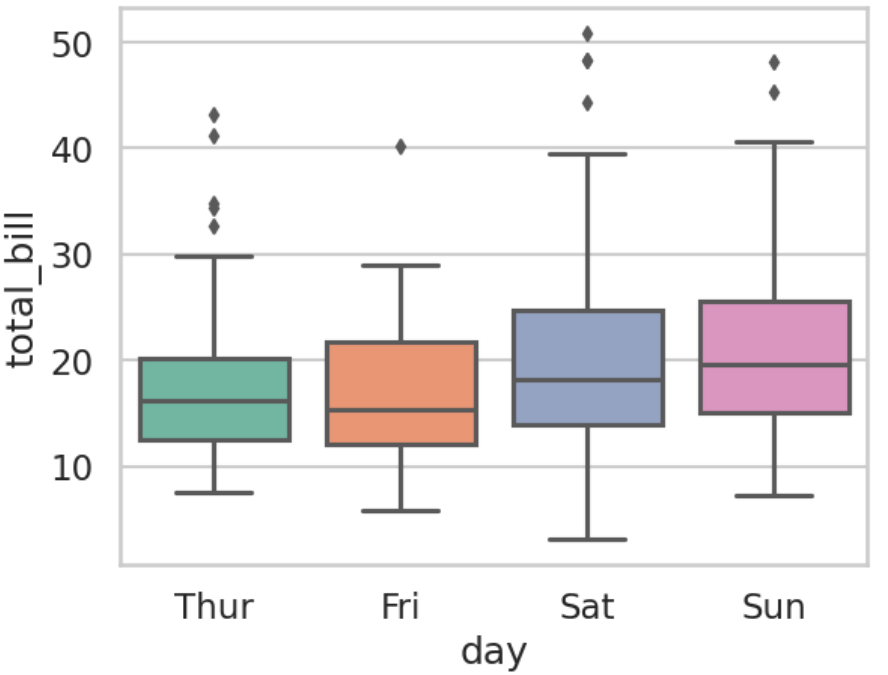

Figure 9: Image generated by the provided code.

This is helpful when you want to apply a different style for a specific plot without altering the global theme.

Summary Table

Function	Purpose
`set_theme()`	Set the overall style and palette

Function	Purpose
`set_style()`	Change the background and grid appearance
`set_palette()`	Choose or define color palettes
`set_context()`	Adjust plot size and elements for different outputs
`with sns.axes_style()`	Temporarily apply a style for a specific plot

Tips for Choosing Style, Palette, and Context

- For **papers**: use `style="whitegrid"`, `context="paper"`, and sequential or diverging palettes.
- For **notebooks**: `style="darkgrid"` and `context="notebook"` work well for quick explorations.
- For **presentations**: increase `font_scale` and use `context="talk"` or `"poster"`.
- Choose palettes that are:
  - Colorblind-friendly.
  - Consistent with the story you want to tell.
  - Not overloaded with too many colors.

In the next chapter, you will learn how to add **Annotations and Details** to make your plots more informative and visually appealing.

Annotations and Details

Introduction

Annotations and small details often make the difference between a simple plot and a great plot. They allow you to:

- Highlight important findings.
- Improve readability.
- Guide the viewer to the key messages.
- Make plots suitable for presentations and publications.

In this chapter, you will learn how to:

- Customize axis labels, titles, legends.
- Add text annotations and markers.
- Draw reference lines.
- Control ticks and grid appearance.

Titles, Axis Labels and Legends

Seaborn integrates smoothly with Matplotlib, so you can easily adjust the basic components of your plots.

Adding title and axis labels

```
sns.scatterplot(data=tips, x="total_bill", y="tip")
plt.title("Tips vs Total Bill")
plt.xlabel("Total Bill ($)")
plt.ylabel("Tip ($)")
plt.show()
```

Explanation:

Always label your axes and give your plot a clear title.

Controlling the legend

```
sns.scatterplot(data=tips,
                x="total_bill", y="tip", hue="sex")
plt.legend(title="Gender", loc="upper left")
plt.show()
```

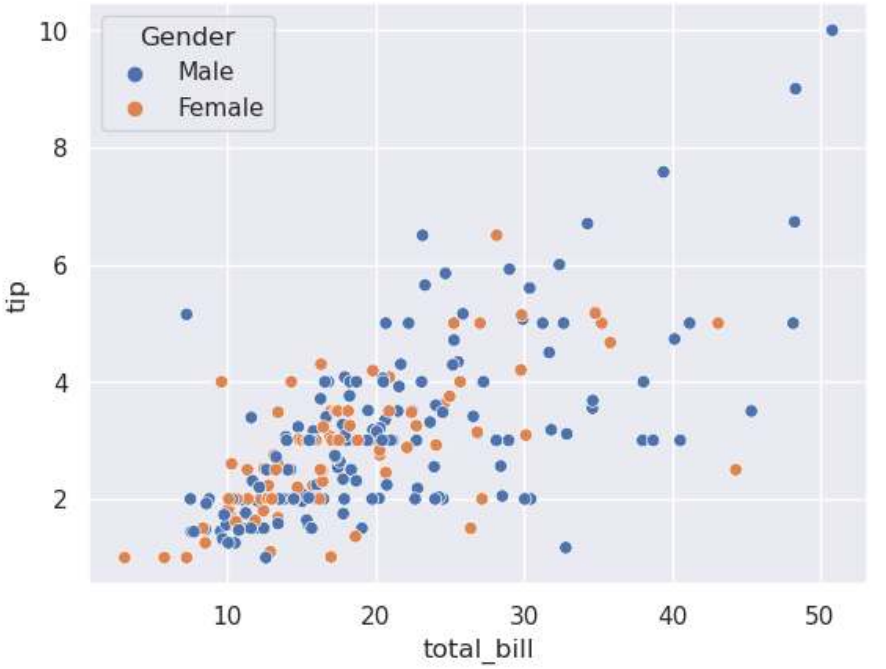

Figure 1: Image generated by the provided code.

Explanation:

You can control the legend position and title using `plt.legend()`.

Adding Text Annotations

You can use `plt.text()` to insert text anywhere in your plot.

Example:

```
sns.scatterplot(data=tips, x="total_bill", y="tip")
plt.text(40, 8, "High tip here!", fontsize=12, color="red
    ")
plt.show()
```

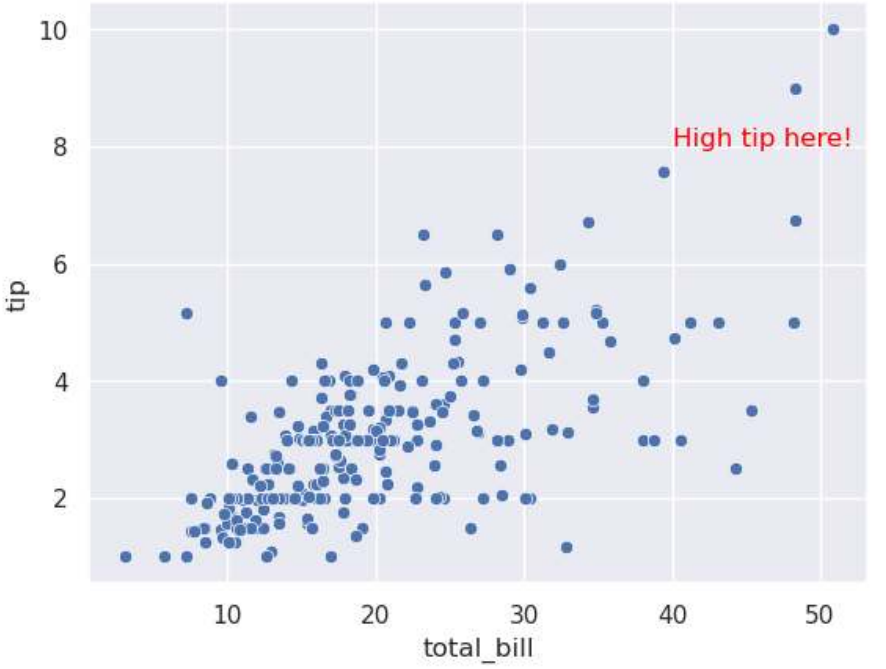

Figure 2: Image generated by the provided code.

Explanation:

Use annotations to highlight specific points or regions.

Highlighting Points

You can also plot specific points with different aesthetics.

Example:

```
sns.scatterplot(data=tips, x="total_bill", y="tip")
plt.scatter(7.4, 5.15, s=200,
            facecolors='none',
            edgecolors='red',
            linewidths=2)
plt.text(7.4, 5.65, "Outlier?", color="red")
plt.show()
```

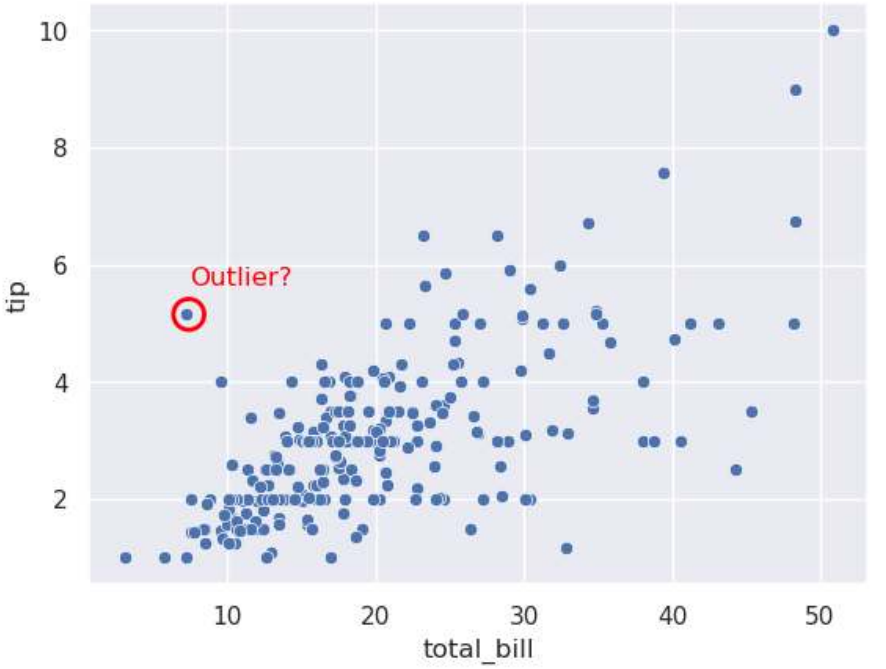

Figure 3: Image generated by the provided code.

Explanation:

This technique is often used to point out outliers or interesting data points.

Adding Reference Lines

Horizontal and vertical lines

```
sns.scatterplot(data=tips, x="total_bill", y="tip")
plt.axhline(5, linestyle="--", color="gray")
plt.axvline(30, linestyle="--", color="gray")
plt.show()
```

- `axhline()` draws a horizontal line.
- `axvline()` draws a vertical line.

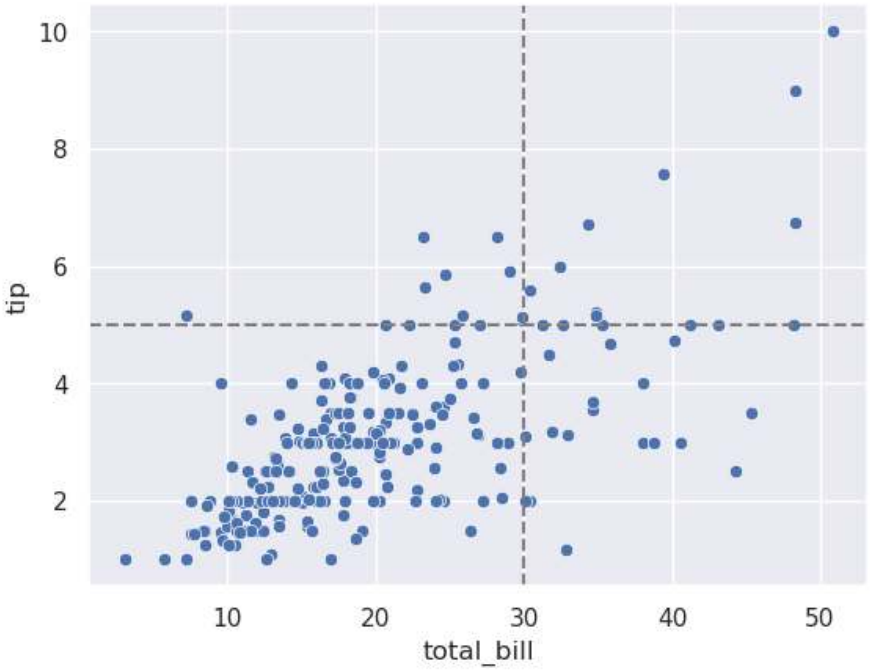

Figure 4: Image generated by the provided code.

Diagonal or custom lines with `plot()`

```
sns.scatterplot(data=tips, x="total_bill", y="tip")
```

```
plt.plot([0, 50], [0, 10], linestyle="--", color="purple"
    )
plt.show()
```

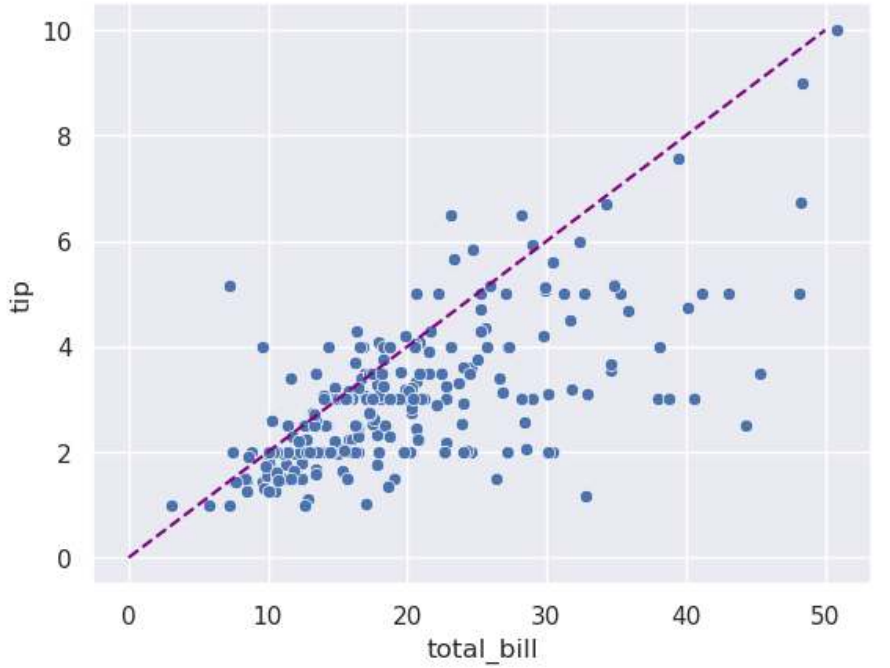

Figure 5: Image generated by the provided code.

Adding Text Boxes

You can create text with background using `bbox`.

```
sns.scatterplot(data=tips, x="total_bill", y="tip")
plt.text(35, 2, "Note:\nLow tips zone",
         fontsize=10,
```

```
        color = "white",
        bbox=dict(boxstyle="round,pad=0.3",
                  edgecolor="black",
                  facecolor="salmon"))
plt.show()
```

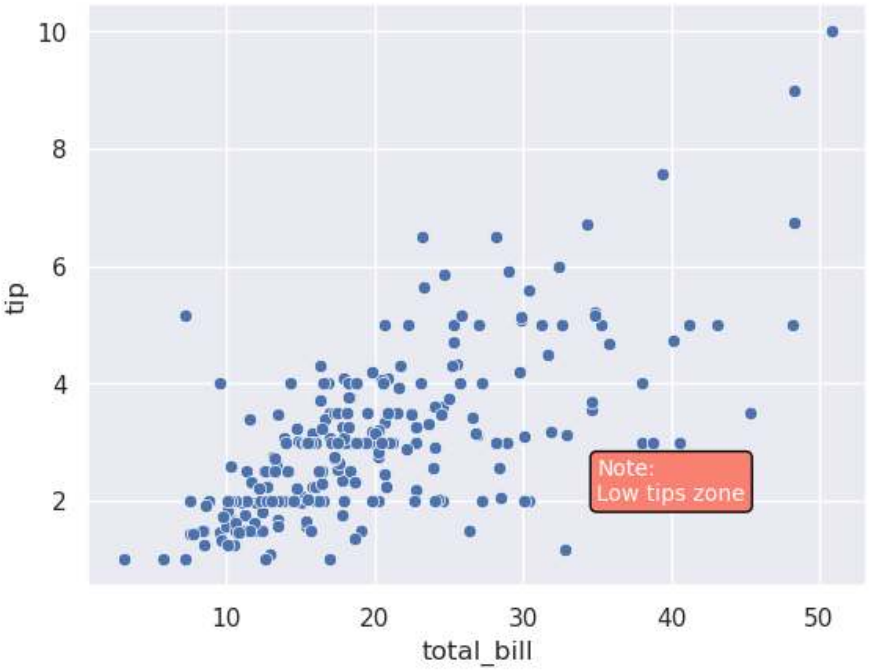

Figure 6: Image generated by the provided code.

Customizing Ticks and Grids

Adjusting ticks

```
sns.scatterplot(data=tips, x="total_bill", y="tip")
plt.xticks([10, 20, 30, 40, 50])
plt.yticks([2, 4, 6, 8, 10])
plt.show()
```

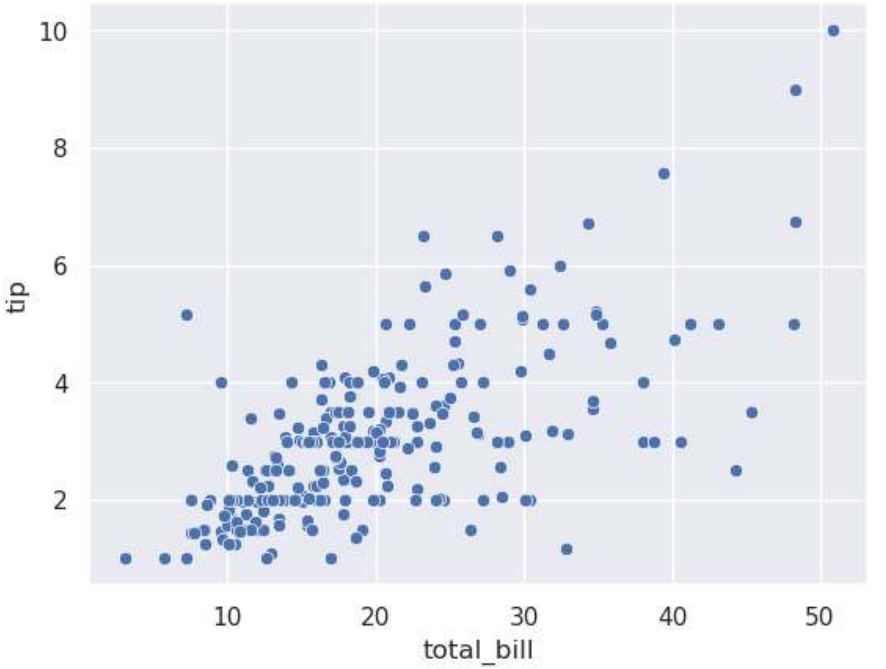

Figure 7: Image generated by the provided code.

Adjusting grid

```
sns.set_style("whitegrid")
sns.scatterplot(data=tips, x="total_bill", y="tip")
plt.grid(True, linestyle="--", linewidth=0.5)
plt.show()
```

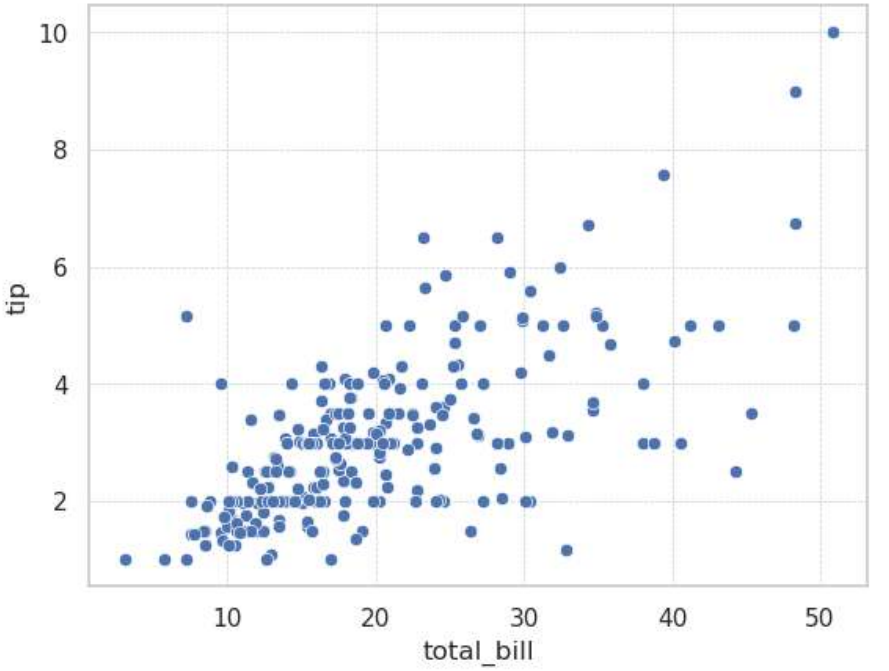

Figure 8: Image generated by the provided code.

Combining All Details

```
sns.scatterplot(data=tips,
                x="total_bill", y="tip", hue="sex")
plt.title("Tips vs Total Bill by Gender")
plt.xlabel("Total Bill ($)")
plt.ylabel("Tip ($)")
plt.axhline(5, linestyle="--", color="gray")
plt.axvline(30, linestyle="--", color="gray")
plt.text(40, 8, "High tip zone", fontsize=12, color="red"
    )
```

```
plt.legend(title="Gender", loc="upper left")
plt.grid(True, linestyle="--", linewidth=0.5)
plt.show()
```

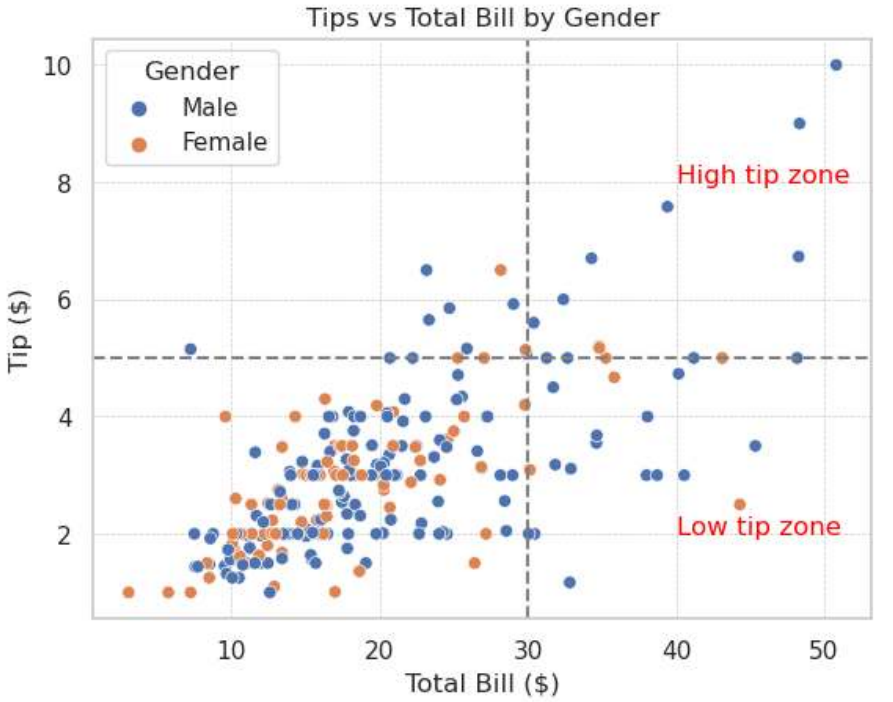

Figure 9: Image generated by the provided code.

This example puts together all the concepts covered in this chapter.

Summary Table

Tool	Purpose
`plt.title()`, `plt.xlabel()`, `plt.ylabel()`	Set plot title and axis labels
`plt.legend()`	Customize legend
`plt.text()`	Add text annotations
`plt.scatter()`	Highlight specific points
`plt.axhline()`, `plt.axvline()`	Add reference lines
`plt.grid()`	Control grid appearance
`plt.xticks()`, `plt.yticks()`	Customize tick marks

In the next chapter, we will learn how to integrate **Seaborn with Matplotlib**, giving you full control over your figures when you need advanced customizations.

Seaborn + Matplotlib

Introduction

Throughout this book, we have focused on using Seaborn to create beautiful and informative visualizations. One of the great advantages of Seaborn is that it is built **on top of Matplotlib**, which means you are never limited to Seaborn's default functionality. In fact, combining Seaborn and Matplotlib is the key to producing fully customized plots.

While Seaborn provides high-level plotting functions with smart defaults and easy-to-use interfaces, Matplotlib gives you **low-level access** to:

- Fine-tune the layout and design of your plots.
- Control elements like axes, legends, annotations, and tick marks.
- Build complex figure compositions (e.g., custom multi-panel layouts).

In other words, **Seaborn makes it easy to create beautiful statistical plots**, and **Matplotlib allows you to make them exactly the way you want**.

In this chapter, you will learn how to:

- Access and manipulate the underlying Matplotlib axes and figures from Seaborn plots.
- Combine multiple plots into custom layouts.
- Add annotations, arrows, reference lines, and other fine details.
- Control legend placement and figure sizes for different use cases.

Mastering this combination will give you full control over your visualizations, whether you are preparing:

- A quick exploratory plot.
- A report figure.
- A slide for a presentation.
- A publication-quality figure.

Let's now explore how Seaborn and Matplotlib work together seamlessly.

Accessing Axes and Figures

Example:

```
ax = sns.scatterplot(data=tips, x="total_bill", y="tip")
ax.set_title("Scatterplot with Matplotlib Title")
ax.set_xlabel("Total Bill ($)")
ax.set_ylabel("Tip ($)")
plt.show()
```

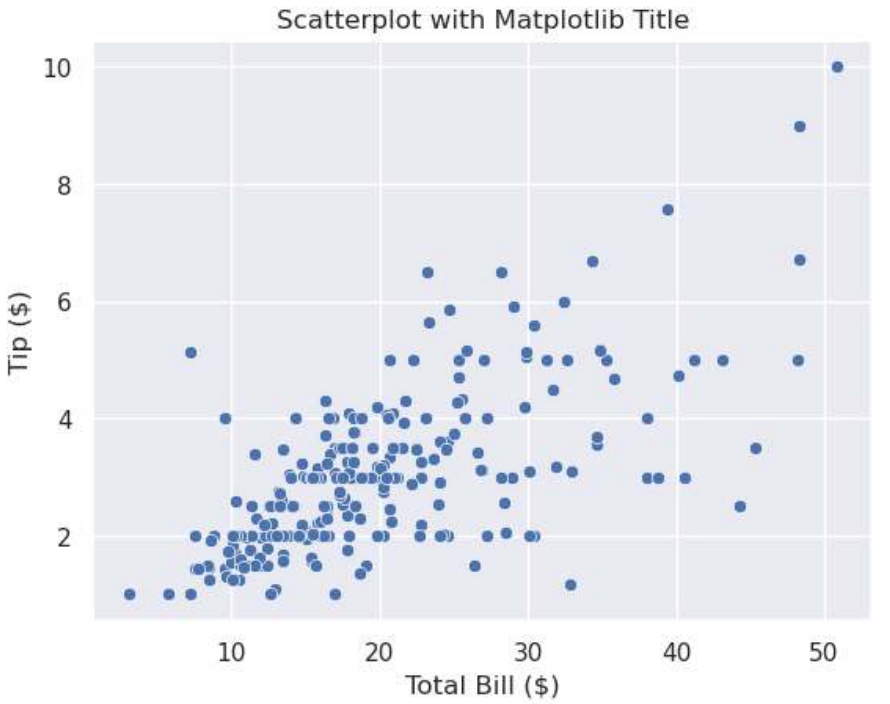

Figure 1: Image generated by the provided code.

Explanation:

- Seaborn returns the `AxesSubplot` object directly.
- You can use `set_title()`, `set_xlabel()`, `set_ylabel()` as you would in pure Matplotlib.

Customizing Tick Labels

```
ax = sns.boxplot(data=tips, x="day", y="total_bill")
ax.set_xticklabels(["Thu", "Fri", "Sat", "Sun"], rotation
    =45)
plt.show()
```

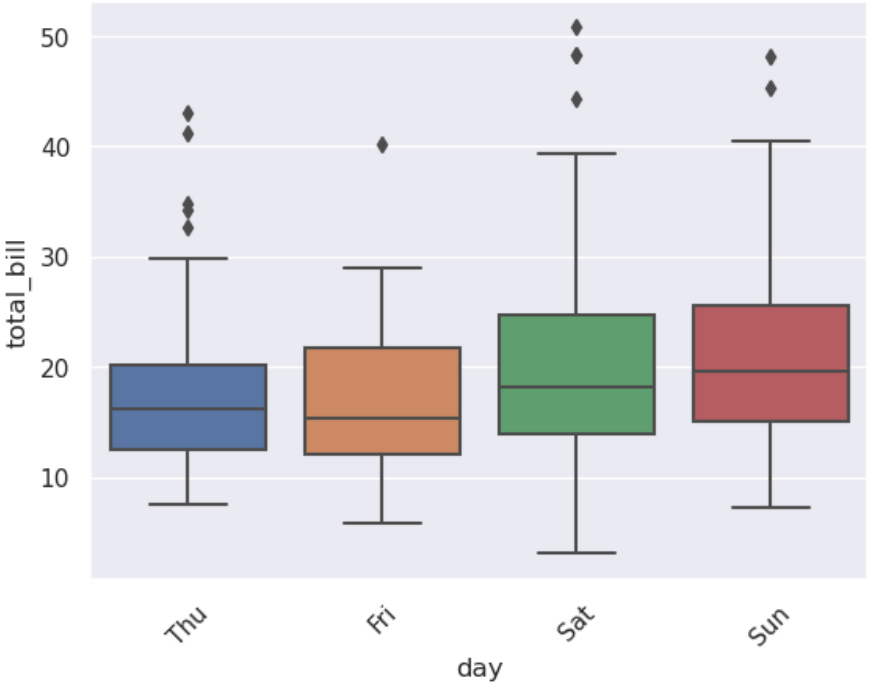

Figure 2: Image generated by the provided code.

Explanation:

Customize tick labels just like in Matplotlib.

Adjusting Plot Spacing

When using multiple plots, it is common to adjust spacing manually.

```
fig, axes = plt.subplots(1, 2, figsize=(10, 4))

sns.histplot(data=tips, x="total_bill", ax=axes[0])
sns.boxplot(data=tips, x="day", y="total_bill", ax=axes
    [1])

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

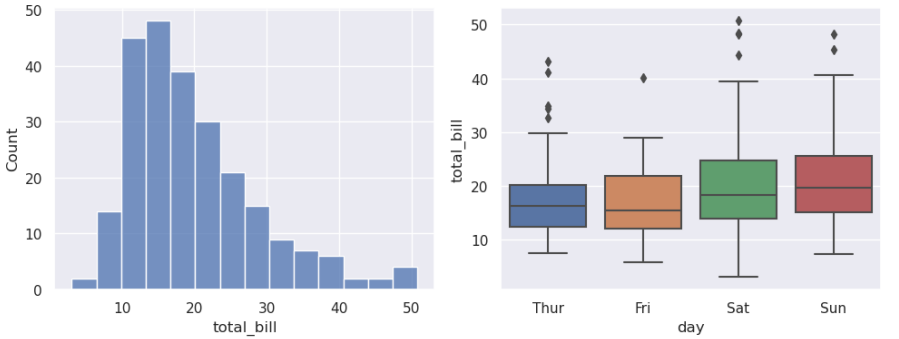

Figure 3: Image generated by the provided code.

Explanation:

- `plt.subplots()` creates multiple Matplotlib axes.
- Pass each `ax` to Seaborn's `ax=` argument.
- `tight_layout()` improves spacing automatically.

Multiple Seaborn Plots in One Figure

Example:

```
fig, axes = plt.subplots(2, 2, figsize=(12, 8))

sns.histplot(data=tips,
             x="total_bill",
             ax=axes[0, 0])
sns.boxplot(data=tips,
            x="day", y="total_bill",
            ax=axes[0, 1])
sns.scatterplot(data=tips,
                x="total_bill", y="tip", hue="sex",
                ax=axes[1, 0])
sns.violinplot(data=tips,
               x="day", y="total_bill", hue="sex",
               split=True, ax=axes[1, 1])

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

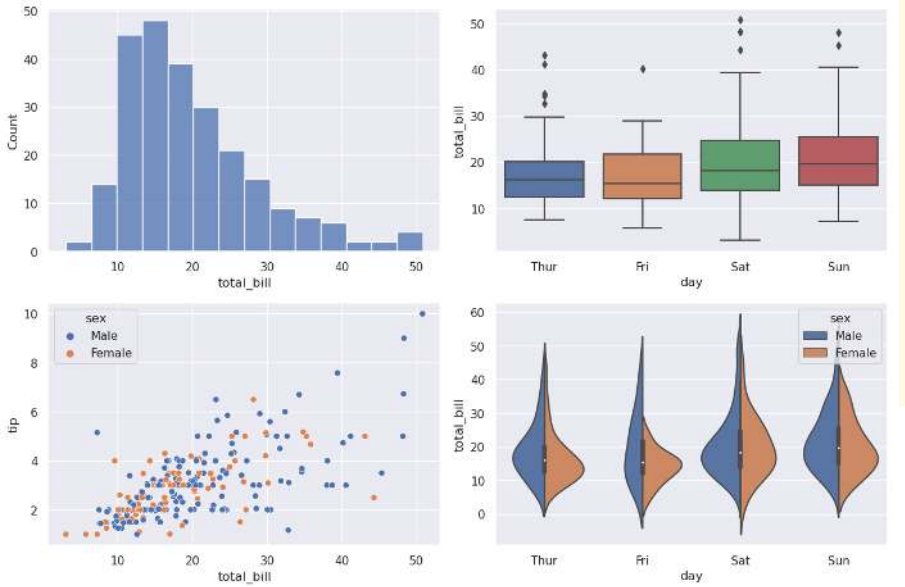

Figure 4: Image generated by the provided code.

This is extremely useful for:

- Creating custom multi-plot layouts.
- Preparing figures for papers and reports.

Combining Seaborn and Matplotlib Elements

You can add Matplotlib annotations, arrows, and shapes on top of Seaborn plots.

```
ax = sns.scatterplot(data=tips, x="total_bill", y="tip")

# Add Matplotlib elements
ax.axhline(5, linestyle="--", color="red")
```

```
ax.text(40, 8, "High tips zone", fontsize=12, color="red")
ax.annotate("Outlier?",
            xy=(7.4, 5.15),
            xytext=(8.9, 6.65),
            arrowprops=dict(facecolor='black',
                            arrowstyle="->"))

plt.show()
```

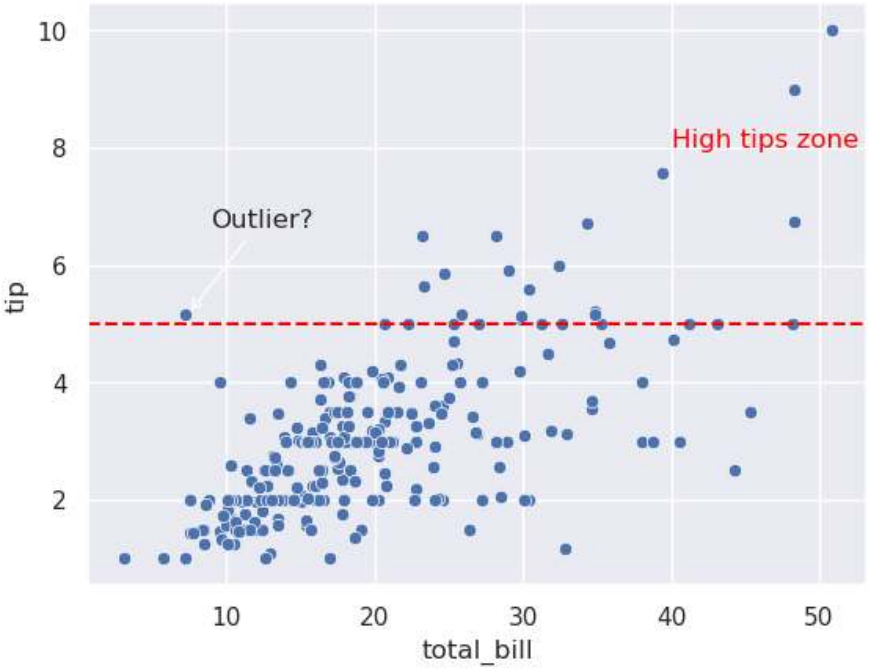

Figure 5: Image generated by the provided code.

Explanation:

- `axhline()`, `text()`, `annotate()` are from Matplotlib but work seamlessly on Seaborn axes.

Fine-Tuning Legends

Moving the legend outside

```
ax = sns.scatterplot(data=tips,
                     x="total_bill", y="tip", hue="sex")
ax.legend(loc="center left", bbox_to_anchor=(1, 0.5))
plt.tight_layout()
plt.show()
```

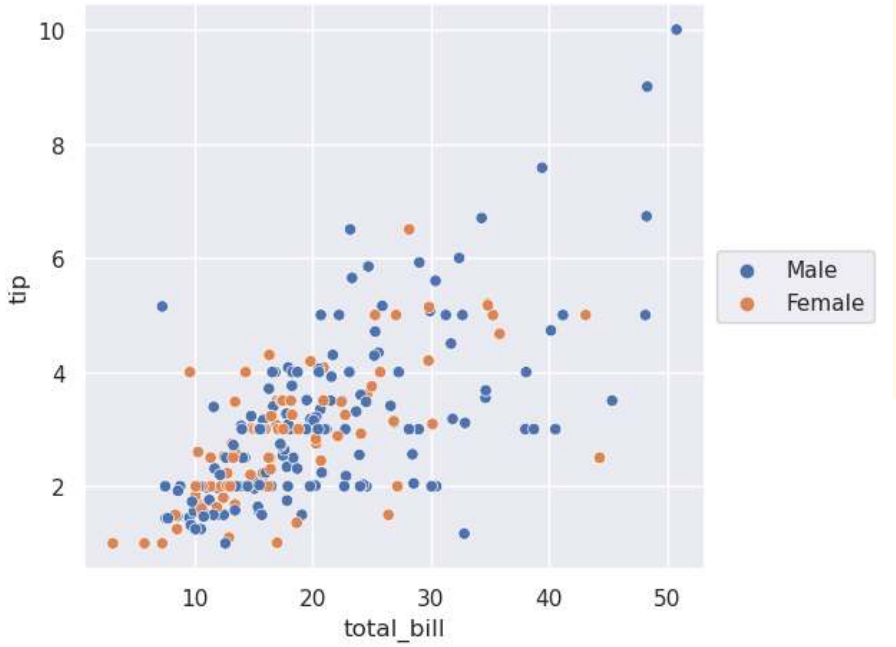

Figure 6: Image generated by the provided code.

This is often necessary when plots are tight or when preparing figures for papers.

Setting Figure Size

Although Seaborn automatically adapts to figure size, you may want full control.

Example:

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

```
sns.boxplot(data=tips, x="day", y="total_bill")
plt.show()
```

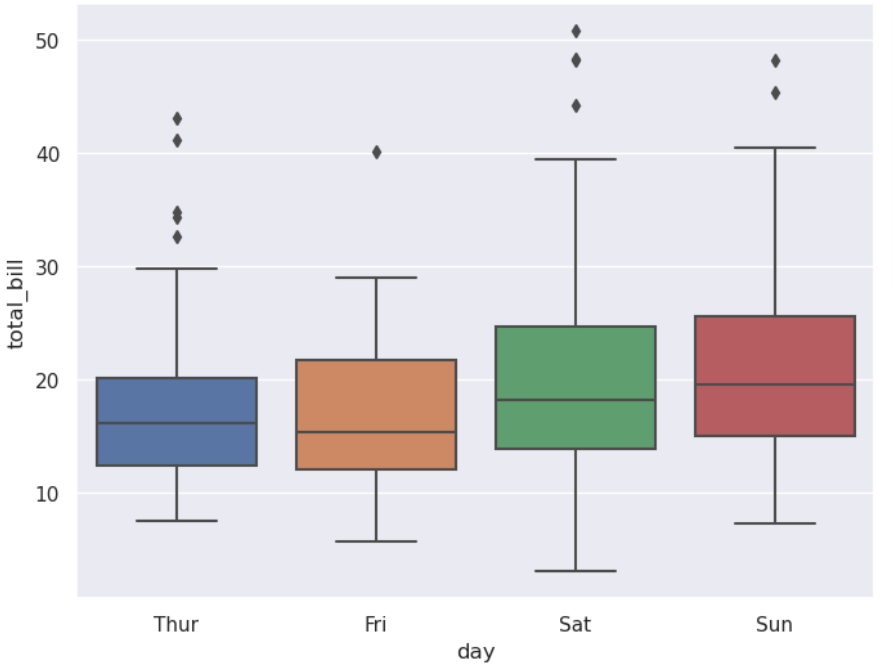

Figure 7: Image generated by the provided code.

Alternatively, use `subplots()` and control individual axes sizes if needed.

Summary Tips

- Use Seaborn for plotting and Matplotlib for **fine-tuning**.
- `plt.subplots()` + `ax=axes[...]` gives you full control over layouts.

- You can freely combine Matplotlib elements like arrows, text, annotations with Seaborn plots.
- Always use `tight_layout()` when building multi-plot figures.

Summary Table

Technique	Purpose
`set_title()`, `set_xlabel()`	Control axis labels and titles
`set_xticklabels()`	Customize tick labels
`plt.subplots()`	Create custom layouts
`tight_layout()`	Fix spacing automatically
`axhline()`, `annotate()`, `text()`	Add Matplotlib elements on top of Seaborn plots
`legend()`	Control legend position and appearance
`figsize`	Set figure size manually

In the next chapter, we will apply everything learned so far in **Case Studies**, where we will produce full, high-quality visualizations for real datasets.

Case Studies

Introduction

In this chapter, we will put everything together by solving realistic problems using Seaborn. We will:

- Explore datasets visually.
- Combine multiple plotting techniques.
- Customize plots for clear communication.
- Prepare publication-quality figures.

By the end of this chapter, you will have a complete workflow, from data exploration to fully customized visualization.

Case Study 1: Exploring the Penguins Dataset

The `penguins` dataset is often used to study relationships between morphological measurements of penguins across species.

Initial Exploration

```
sns.pairplot(penguins.dropna(), hue="species")
plt.show()
```

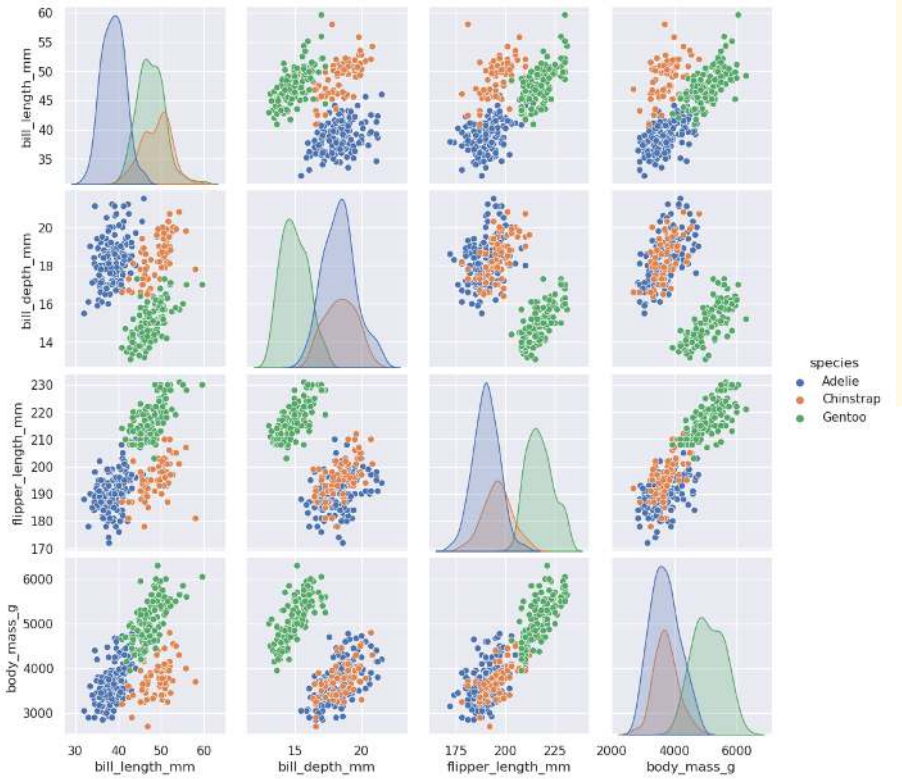

Figure 1: Image generated by the provided code.

Observation:

- We can identify clusters of species.
- Strong relationships appear between `flipper_length_mm`, `bill_length_mm`, and `body_mass_g`.

Advanced Scatterplot with Regression

```
sns.lmplot(
    data=penguins,
    x="flipper_length_mm",
    y="body_mass_g",
    hue="species",
    height=6,
    aspect=1.2,
    markers=["o", "s", "D"]
)
plt.title("Flipper Length vs Body Mass by Species")
plt.xlabel("Flipper Length (mm)")
plt.ylabel("Body Mass (g)")
plt.tight_layout()
plt.show()
```

- Regression lines highlight species differences.
- Markers differentiate species visually.

Figure 2: Image generated by the provided code.

Case Study 2: Tips Dataset - Insights for a Restaurant Manager

The `tips` dataset could represent real data from a restaurant manager interested in understanding customer tipping behavior.

Total Bill vs Tip Relationship

```
sns.scatterplot(data=tips,
                x="total_bill", y="tip", hue="day",
```

```
                  style="sex")
plt.title("Tip Amount vs Total Bill")
plt.xlabel("Total Bill ($)")
plt.ylabel("Tip ($)")
plt.legend(title="Day / Sex")
plt.grid(True, linestyle="--", linewidth=0.5)
plt.tight_layout()
plt.show()
```

Observation:

- Higher bills tend to receive higher tips.
- The relationship varies slightly across days and customer gender.

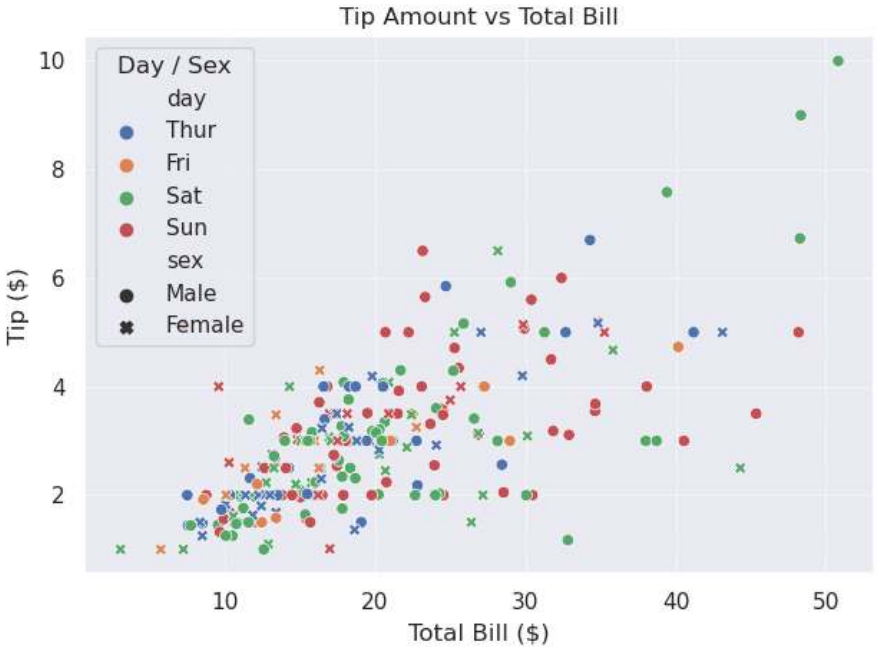

Figure 3: Image generated by the provided code.

Average Tips by Day and Smoker Status

```
sns.catplot(data=tips, x="day", y="tip", hue="smoker",
            kind="box", height=5, aspect=1.5)
plt.title("Tips by Day and Smoker Status")
plt.tight_layout()
plt.show()
```

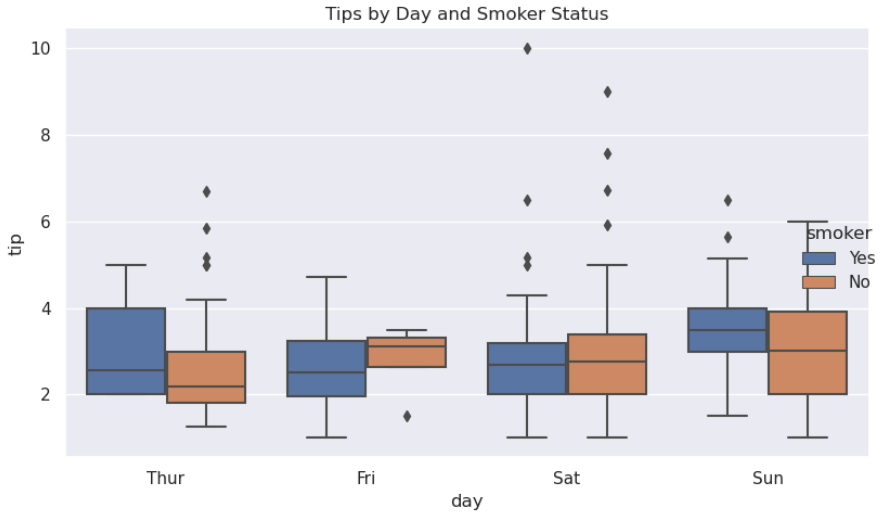

Figure 4: Image generated by the provided code.

- Smokers and non-smokers show different tipping patterns depending on the day.

Case Study 3: Time Trends in the Flights Dataset

This dataset shows the monthly number of passengers on an airline over several years.

Heatmap of Passenger Counts

```
flights_pivot = flights.pivot(index="month",
                              columns="year",
                              values="passengers")

sns.heatmap(flights_pivot, annot=True, fmt="d", cmap="
    YlGnBu")
plt.title("Monthly Passenger Counts (1949-1960)")
plt.xlabel("Year")
plt.ylabel("Month")
plt.tight_layout()
plt.show()
```

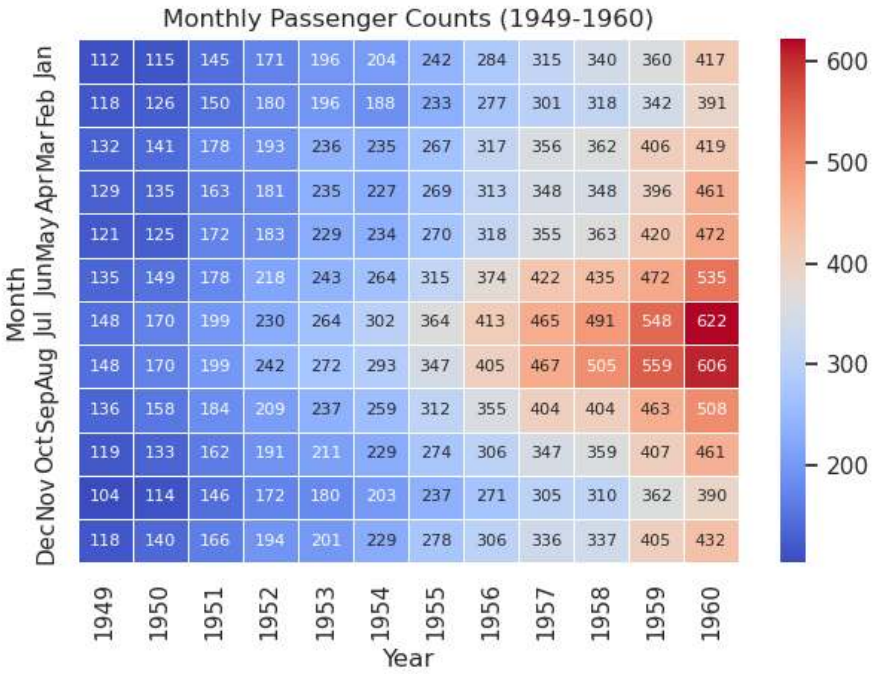

Figure 5: Image generated by the provided code.

- Clear seasonality is visible.
- Passenger counts increase year over year.

Clustered Heatmap

```
sns.clustermap(flights_pivot, cmap="coolwarm")
plt.show()
```

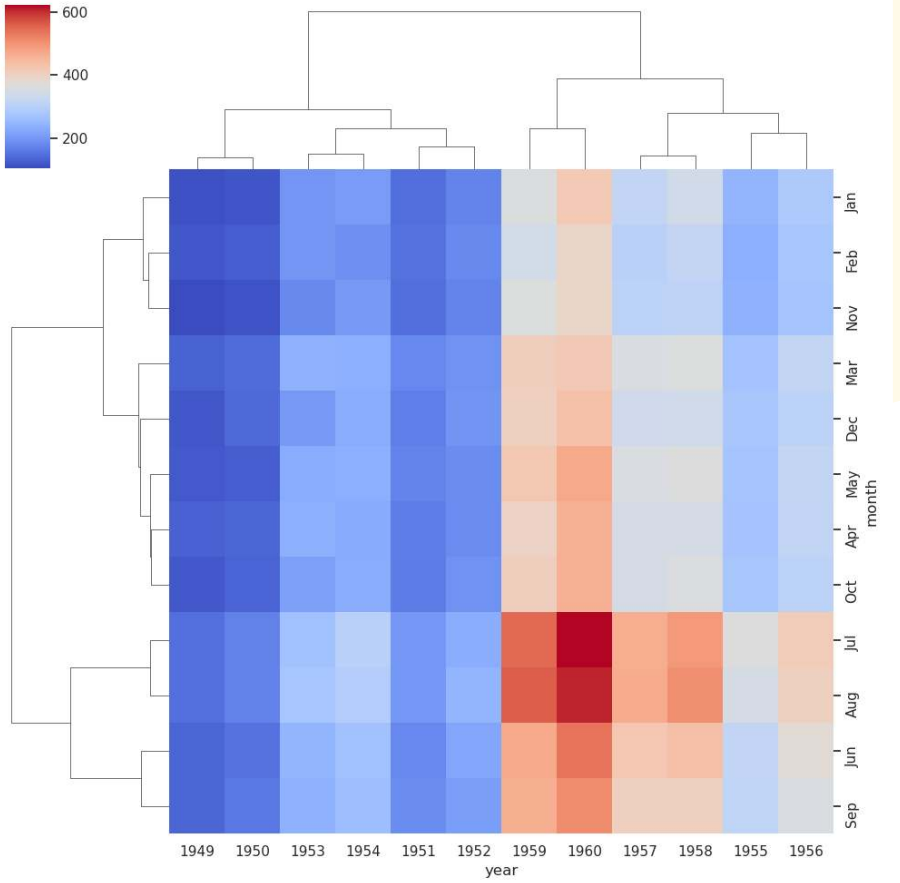

Figure 6: Image generated by the provided code.

- Groups similar months and years automatically.
- Highlights potential patterns and similarities.

Case Study 4: Creating a Publication-Ready Composite Figure

Let's combine several plots into a single figure as you would for a report or paper.

```
fig, axes = plt.subplots(2, 2, figsize=(12, 10))

# Scatterplot
sns.scatterplot(data=tips,
                x="total_bill", y="tip", hue="day",
                ax=axes[0, 0])
axes[0, 0].set_title("Total Bill vs Tip")

# Boxplot
sns.boxplot(data=tips,
            x="day", y="tip", hue="smoker",
            ax=axes[0, 1])
axes[0, 1].set_title("Tips by Day and Smoker")

# Regression plot
sns.regplot(data=penguins,
            x="flipper_length_mm", y="body_mass_g",
            ax=axes[1, 0])
axes[1, 0].set_title("Flipper Length vs Body Mass")

# Heatmap
sns.heatmap(flights_pivot,
            annot=True,
            fmt="d",
            cmap="coolwarm",
            linewidths=0.5,
            linecolor="white",
            annot_kws={"size":8},
            ax = axes[1, 1])
axes[1, 1].set_title("Flights Heatmap")

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

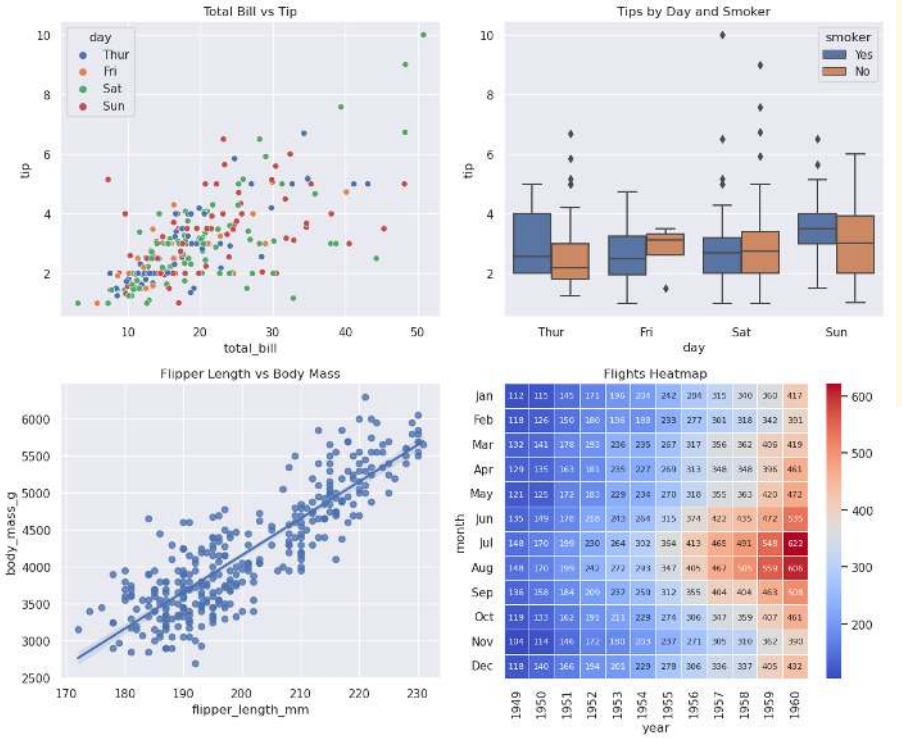

Figure 7: Image generated by the provided code.

Notes

- Combining different Seaborn plots into a **single figure** is a powerful way to communicate multiple aspects of your data.
- `plt.subplots()` + `ax=axes[...]` + Seaborn plots allow for total flexibility.
- Do not forget to adjust `titles`, `legends`, `font sizes`, and `spacing` when preparing figures for reports or publications.

Key Takeaways

- Use Seaborn for its simple but powerful plotting functions.
- Use Matplotlib to customize details and to arrange plots as needed.
- Combine both libraries to create complex, yet clear and informative figures.

In the next chapter, we will summarize useful **Tips and Tricks** to improve your plots even more and avoid common pitfalls.

Tips and Tricks

Introduction

In this chapter, we will share a collection of **practical tips**, **common pitfalls**, and **best practices** that will help you:

- Improve the quality of your plots.
- Save time when working with Seaborn.
- Avoid common mistakes.
- Prepare plots for notebooks, presentations, or publications.

These recommendations are based on real-world experience and are applicable regardless of your level of expertise.

Plotting Tips

Tip 1 — Set the theme at the beginning of your analysis

```
sns.set_theme(style="whitegrid",
              palette="muted",
              context="notebook")
```

Setting a global style avoids inconsistent plots across your notebook or report.

Tip 2 — Adjust figure size early when needed

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

This avoids having to resize plots later or having plots too small when exporting.

Tip 3 — Use `hue`, `style`, and `size` meaningfully

These arguments are powerful but can make plots too crowded if overused. Always ask yourself: *Does adding color or style improve readability?*

Tip 4 — Use `tight_layout()` often

```
plt.tight_layout()
```

It helps automatically fix spacing issues, especially when combining multiple plots.

Customization Tricks

Trick 1 — Combine Seaborn and Matplotlib

Seaborn handles the plot, but Matplotlib is your best ally for annotations, arrows, text boxes, or precise layout adjustments.

```
ax = sns.scatterplot(data=tips, x="total_bill", y="tip")
ax.annotate("Interesting point",
            xy=(40, 8),
            xytext=(30, 9),
            arrowprops=dict(arrowstyle="->"))
```

Trick 2 — Control legend location manually

```
plt.legend(loc="center left", bbox_to_anchor=(1, 0.5))
```

Moving legends outside the plot is often necessary in papers and slides.

Trick 3 — Combine multiple plots in one figure

```
fig, axes = plt.subplots(1, 2, figsize=(12, 5))
sns.boxplot(data=tips, x="day", y="tip", ax=axes[0])
sns.violinplot(data=tips, x="day", y="tip", ax=axes[1])
plt.tight_layout()
plt.show()
```

It is often clearer to show two complementary plots side by side than to overload one.

Interpretation Tips

- Avoid **overplotting**: Don't use too many variables encoded via `hue`, `size`, and `style` simultaneously.
- Always check if the plot helps answer your question.
- Use **color carefully**:
  - Avoid unnecessary colors.
  - Use colorblind-friendly palettes (e.g., `"colorblind"`, `"muted"`, `"Set2"`).
- Customize labels: change `plt.title()`, `plt.xlabel()`, and `plt.ylabel()` to make plots self-explanatory.

Exporting Plots

Saving plots properly

```
plt.savefig("my_plot.png",
            dpi=300,
            bbox_inches="tight")
```

- Always export plots with high resolution (`dpi=300`) if you plan to use them in papers.
- Use `bbox_inches="tight"` to remove extra white space.

Avoid exporting screenshots:

Always export using `savefig()` rather than screenshots to preserve quality and dimensions.

Avoiding Common Mistakes

Mistake	Recommendation
Using default labels	Always add informative titles and axis labels
Overloading with variables	Use `hue`, `style`, or `size` selectively
Ignoring legends	Always adjust or clean your legends
Forgetting to adjust figure size	Use `figsize` or `set_context()` appropriately
Not checking colorblind accessibility	Prefer palettes like `"colorblind"` or `"muted"`

Final Thoughts

Seaborn is designed to help you:

- Produce informative, clean, and beautiful plots with minimal code.
- Combine easily with Matplotlib for advanced control.
- Focus on the story you want to tell through your data.

Visualization is not just about making plots; it's about making **insights visible**.

In the next (and final) chapter, we will provide a list of **References and Further Reading** to help you continue learning and improving your data visualization skills.

References and Further Reading

Official Documentation

The following resources are official and maintained by the Seaborn and Matplotlib developers:

- Seaborn Official Documentation
  https://seaborn.pydata.org/
- Matplotlib Official Documentation
  https://matplotlib.org/stable/index.html
- Pandas Official Documentation
  https://pandas.pydata.org/docs/
- Python Data Science Handbook (Jake VanderPlas)
  https://jakevdp.github.io/PythonDataScienceHandbook/

Recommended Books

- **Data Visualization with Python and Seaborn**
  By: Marc Garcia
  A focused book on producing effective visualizations using Seaborn.
- **Python Data Science Handbook**
  By: Jake VanderPlas

A complete guide to data science with Python, including Matplotlib and Seaborn.

- **Storytelling with Data**
  By: Cole Nussbaumer Knaflic
  Highly recommended to improve your ability to communicate insights through visuals.
- **Fundamentals of Data Visualization**
  By: Claus O. Wilke
  Open-access book explaining good practices in data visualization.
  https://clauswilke.com/dataviz/

Tutorials and Blogs

- **Seaborn Tutorials**
  https://seaborn.pydata.org/tutorial.html
- **DataCamp Seaborn Tutorial**
  https://www.datacamp.com/tutorial/seaborn-python-tutorial
- **Practical Business Python Blog**
  https://pbpython.com/
- **Towards Data Science (Visualization Category)**
  https://towardsdatascience.com/tagged/data-visualization

Color and Style Resources

- **Color Brewer 2** (Colorblind-friendly palettes)
  https://colorbrewer2.org/

- **Adobe Color Wheel** (For creating custom palettes)
  https://color.adobe.com/
- **Seaborn Color Palette Reference**
  https://seaborn.pydata.org/tutorial/color_palettes.html

Recommended Practice

To master data visualization:

1. Practice by reproducing plots from articles, papers, or online tutorials.
2. Analyze figures in scientific publications and think about what works and what could be improved.
3. Experiment with different datasets and explore their relationships visually.
4. Combine Seaborn with Matplotlib when you need full control.

Final Words

Visualization is a key skill for any data scientist, analyst, or researcher. Seaborn provides a gentle learning curve with powerful tools, but always remember:

> "A good plot is not only informative but also tells a story."

Keep exploring, experimenting, and refining your visual storytelling skills!

Thank you for following this guide!


rubió
Metabolomics