Introduction to Data Visualization

Why Data Visualization is Important

Data visualization is the bridge between raw data and human understanding. When done right, it helps:

- Reveal patterns, trends, and correlations in the data.
- Communicate insights clearly to stakeholders.
- Speed up decision-making by simplifying complex datasets.
- Make data storytelling engaging and accessible to all.

> A picture is worth a thousand words

> “The greatest value of a picture is when it forces us to notice what we never expected to see.” – John Tukey

---

Exploratory vs Explanatory Visuals

Exploratory Visualizations

- Purpose: Explore the data, uncover insights, find patterns.
- Audience: You (the data analyst/scientist).
- Example: Pair plots, correlation heatmaps, scatter matrix.

Explanatory Visualizations

- Purpose: Communicate a specific insight or story.
- Audience: Stakeholders, clients, public.
- Example: A bar chart in a presentation showing sales trends.

Aspect	Exploratory	Explanatory
Goal	Find insights	Communicate insights
Audience	Analyst / Data Scientist	Stakeholders / Public
Style	Raw, fast, flexible	Polished, focused, clean

Basic Principles of Good Visualizations

1. **Clarity**
   Avoid clutter. Use labels, legends, and proper axis scales.
2. **Context**
   Always provide context: What is being measured? Over what time frame? In what units?
3. **Focus**
   Highlight the key insight. Use colors and annotations to draw attention.
4. **Storytelling**
   Don’t just show data — tell a story. Guide the viewer through a narrative.
5. **Accessibility**
   Use carefully chosen color palettes that enhance readability for all viewers.

Pro Tip:

> Always ask yourself: “What is the one thing I want the viewer to understand from this visual?”

Introduction to Matplotlib

Today, we'll cover the **basics of Matplotlib** — the most fundamental plotting library in Python. By the end, you'll understand how to make clean and powerful plots, step by step.

What is `matplotlib.pyplot`?

- `matplotlib.pyplot` is a module in Matplotlib — it's like a paintbrush for your data.
- We usually import it as `plt`:

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

> `plt` is just a short alias to save typing!

What is `plt.show()`?

- `plt.show()` is used to **display the plot**.
- Without it, in scripts, you might not see the plot window.

```
plt.plot([1, 2, 3], [4, 5, 6])
plt.show()
```

Interacting with the Plot

When a plot appears, you can:

- Zoom In/Out
- Pan around
- Use arrows to navigate history
- Reset to home
- Save as PNG using the disk icon

> These features are **automatically included** in the plot window!

Real Data Example: Sachin Tendulkar's Runs Over Time

```
years = [1990, 1992, 1994, 1996, 1998, 2000, 2003, 2005, 2007, 2010]
runs =  [500, 700, 1100, 1500, 1800, 1200, 1700, 1300, 900, 1500]
```

```
plt.plot(years, runs)
plt.show()
```

Adding X and Y Labels

```
plt.plot(years, runs)
plt.xlabel("Year")
plt.ylabel("Runs Scored")
plt.title("Sachin Tendulkar's Yearly Runs")
plt.show()
```

Multiple Lines in One Plot

```
kohli = [0, 0, 500, 800, 1100, 1300, 1500, 1800, 1900, 2100]
sehwag = [0, 300, 800, 1200, 1500, 1700, 1600, 1400, 1000, 0]

plt.plot(years, kohli, label="Virat Kohli")
plt.plot(years, sehwag, label="Virender Sehwag")

plt.xlabel("Year")
plt.ylabel("Runs Scored")
plt.title("Performance Comparison")
plt.legend()
plt.show()
```

Why `label` is Better than List in `legend`

Bad practice:

```
plt.plot(years, kohli)
plt.plot(years, sehwag)
plt.legend(["Kohli", "Sehwag"])  # prone to mismatch
```

Better:

```
plt.plot(years, kohli, label="Kohli")
plt.plot(years, sehwag, label="Sehwag")
plt.legend()
```

Using Format Strings

```
plt.plot(years, kohli, 'ro--', label="Kohli")  # red circles with dashed lines
plt.plot(years, sehwag, 'g^:', label="Sehwag")  # green triangles dotted
plt.legend()
```

Color and Line Style Arguments

```
plt.plot(years, kohli, color='orange', linestyle='--', label="Kohli")
plt.plot(years, sehwag, color='green', linestyle='-.', label="Sehwag")
plt.plot(years, runs, color='blue', label="Tendulkar")
plt.legend()
```

Line Width and Layout Tweaks

```
plt.plot(years, kohli, linewidth=3, label="Kohli")
plt.plot(years, sehwag, linewidth=2, label="Sehwag")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Using Styles in Matplotlib

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

Try a few:

```
plt.style.use("ggplot")
# or
plt.style.use("seaborn-v0_8-bright")
```

XKCD Comic Style

```
with plt.xkcd():
    plt.plot(years, kohli, label="Kohli")
    plt.plot(years, sehwag, label="Sehwag")
    plt.title("Epic Battle of the Batsmen")
    plt.legend()
    plt.show()
```

Visualizing Tons of Data – What Crowded Looks Like

```
import numpy as np

for i in range(50):
    plt.plot(np.random.rand(100), linewidth=1)

plt.title("Too Much Data Can Be Confusing!")
plt.grid(True)
plt.tight_layout()
plt.show()
```

Final Tips for Beginners

- Always start with simple plots
- Add labels and legends early
- Use `plt.grid()` and `plt.tight_layout()` to improve readability
- Try different styles to find what works for your use case

Assignment:
Create a plot comparing **Kohli**, **Rohit Sharma**, and **Sehwag** across 10 years of hypothetical runs.
Use:

- Labels
- Legends
- Colors
- Line styles
- One custom style

Matplotlib Bar Charts

Bar charts are used to **compare quantities** across categories. They are easy to read and powerful for visual analysis.

We'll use **Sachin Tendulkar's yearly run data**, then learn how to create grouped bar charts and horizontal bar charts with examples.

Sachin's Yearly Runs – Vertical Bar Chart

The Data

```
import matplotlib.pyplot as plt

years = [1990, 1992, 1994, 1996, 1998, 2000, 2003, 2005, 2007, 2010]
runs =  [500, 700, 1100, 1500, 1800, 1200, 1700, 1300, 900, 1500]
```

Basic Bar Plot

```
plt.bar(years, runs)
plt.xlabel("Year")
plt.ylabel("Runs Scored")
plt.title("Sachin Tendulkar's Yearly Runs")
plt.show()
```

Setting Bar Width & Side-by-Side Bar Charts

Let’s compare **Sachin**, **Sehwag**, and **Kohli** side-by-side for the same years.

```
import numpy as np

sachin = [500, 700, 1100, 1500, 1800, 1200, 1700, 1300, 900, 1500]
sehwag = [0, 200, 900, 1400, 1600, 1800, 1500, 1100, 800, 0]
kohli  = [0, 0, 500, 800, 1100, 1300, 1500, 1800, 1900, 2100]

x = np.arange(len(years))  # index positions
width = 0.25
```

The value of width ranges from 0 to 1 (default 0.8); it can go above 1, but bars will start to overlap.

Plotting Side-by-Side Bars

```
plt.bar(x - width, sachin, width=width, label="Sachin")
plt.bar(x, sehwag, width=width, label="Sehwag")
plt.bar(x + width, kohli, width=width, label="Kohli")

plt.xlabel("Year")
plt.ylabel("Runs")
plt.title("Run Comparison")
plt.xticks(x, years)  # Show actual year instead of 0,1,2,...
plt.legend()
plt.tight_layout()
plt.show()
```

Why Use `xticks()`?

By default, `plt.bar()` uses numeric x-values (0, 1, 2, ...).
We use `plt.xticks()` to set the correct category labels like years or names.

Horizontal Bar Charts with `barh()`

Let’s compare **total runs scored in the first 5 years** by different players.

```
players = ["Sachin", "Sehwag", "Kohli", "Yuvraj"]
runs_5yrs = [500+700+1100+1500+1800, 0+200+900+1400+1600, 0+0+500+800+1100,
300+600+800+1100+900]
```

Plotting with `barh()`

```
plt.barh(players, runs_5yrs, color="skyblue")
plt.xlabel("Total Runs in First 5 Years")
plt.title("First 5-Year Performance of Indian Batsmen")
plt.tight_layout()
plt.show()
```

Why Switch `x` and `y`?

- In `bar()` → `x = categories`, `y = values`
- In `barh()` → `y = categories`, `x = values`

> Horizontal bars help when category names are long or when you want to emphasize comparisons from left to right.

Adding Value Labels with `plt.text()`

You can use `plt.text()` to display values **on top of bars**, making your chart easier to read. It's especially useful in presentations and reports.

```
import matplotlib.pyplot as plt

players = ["Sachin", "Sehwag", "Kohli"]
runs = [1500, 1200, 1800]

plt.bar(players, runs, color="skyblue")

# Add labels on top of bars
for i in range(len(players)):
    plt.text(i, runs[i] + 50, str(runs[i]), ha='center')
plt.ylabel("Runs")
plt.title("Runs Scored by Players")
plt.tight_layout()
plt.show()
```

Tip: Use `ha='center'` to center the text and add a small offset (`+50` here) to avoid overlap with the bar.

Summary

Feature	Use
`plt.bar()`	Vertical bars for categorical comparison
`plt.barh()`	Horizontal bars (great for long labels)
`width=`	Control thickness/spacing of bars
`np.arange()`	Helps to align multiple bars side by side
`plt.xticks()`	Replaces index numbers with real labels
`plt.tight_layout()`	Prevents labels from overlapping

Assignment:
Create a side-by-side bar chart comparing runs of Sachin, Kohli, and Sehwag over 5 selected years.
Then create a horizontal bar chart showing total runs scored in their debut 5 years.

Matplotlib Pie Charts

Pie charts are used to show **part-to-whole relationships**. They're visually appealing but best used with **fewer categories**, as too many slices can get cluttered and hard to interpret.

Setting a Style

```
import matplotlib.pyplot as plt

plt.style.use("ggplot")  # Choose any style you like
```

Basic Pie Chart Example

```
# Data
labels = ["Sachin", "Sehwag", "Kohli", "Yuvraj"]
runs = [18000, 8000, 12000, 9500]
```

```
plt.title("Career Runs of Indian Batsmen")
plt.pie(runs, labels=labels)
plt.show()
```

Custom Colors

You can use color names or hex codes.

```
colors = ['#ff9999','#66b3ff','#99ff99','#ffcc99']
plt.pie(runs, labels=labels, colors=colors)
plt.show()
```

Add Edges and Style Slices with `wedgeprops`

You can customize the look of the slices using `wedgeprops`.

```
plt.pie(
    runs,
    labels=labels,
    colors=colors,
    wedgeprops={'edgecolor': 'black', 'linewidth': 2, 'linestyle': '--'}
)
plt.show()
```

Tip: You can Google "matplotlib wedgeprops" for more customization options.

Visit this documentation page for more

Highlight a Slice Using `explode`

Use `explode` to **pull out** one or more slices.

```
explode = [0.1, 0, 0, 0]  # Only highlight Sachin's slice

plt.pie(runs, labels=labels, explode=explode, colors=colors)
plt.title("Exploded Pie Example")
plt.show()
```

Add Shadows for a 3D Feel

```
plt.pie(
    runs,
    labels=labels,
    explode=explode,
    colors=colors,
    shadow=True  # adds a 3D-like shadow
)
plt.show()
```

Start Angle

Use `startangle` to rotate the pie for better alignment.

```
plt.pie(
    runs,
    labels=labels,
    explode=explode,
    colors=colors,
    shadow=True,
    startangle=140
)
plt.show()
```

Show Percentages with `autopct`

```
plt.pie(
    runs,
    labels=labels,
    autopct='%1.1f%%',
    startangle=140
)
plt.title("Career Run Share")
plt.show()
```

Crowded Pie Chart – Why to Avoid

```
# Too many categories
languages = ['Python', 'Java', 'C++', 'JavaScript', 'C#', 'Ruby', 'Go', 'Rust',
'Swift', 'PHP']
usage = [30, 20, 10, 10, 7, 5, 4, 3, 2, 1]

plt.pie(usage, labels=languages, autopct='%1.1f%%', startangle=90)
plt.title("Programming Language Usage (Crowded Example)")
plt.show()
```

Why avoid it?

- Hard to compare slice sizes visually
- Cluttered and confusing
- No clear insight
  Better alternatives: bar charts or horizontal bar charts

Summary of Useful Pie Chart Parameters

Parameter	Use
`labels`	Label each slice
`colors`	Customize slice colors (names or hex)
`explode`	Pull out slices for emphasis
`shadow`	Adds depth-like shadow
`startangle`	Rotates pie to start from a different angle
`autopct`	Shows percentage text on slices
`wedgeprops`	Customize slice edge, fill, width, etc.

Assignment:
Create a pie chart showing the market share of mobile OS (Android, iOS, others). Then recreate the same using a horizontal bar chart and observe which is easier to understand.

Stack Plots in Matplotlib

Let's Start with a Pie Chart

Before we understand what a stack plot is, let’s visualize some simple data

Imagine you surveyed how a group of students spends their after-school time:

```
import matplotlib.pyplot as plt

activities = ['Studying', 'Playing', 'Watching TV', 'Sleeping']
time_spent = [3, 2, 2, 5]  # hours in a day

colors = ['skyblue', 'lightgreen', 'gold', 'lightcoral']

plt.figure(figsize=(6,6))
plt.pie(time_spent, labels=activities, colors=colors, autopct='%1.1f%%',
startangle=90)
plt.title("After School Activities")
plt.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.
plt.show()
```

Interpretation:

- **Studying**: 3 hours
- **Playing**: 2 hours
- **Watching TV**: 2 hours
- **Sleeping**: 5 hours

This pie chart shows how a single student spends time **in one day**.

What is a Stack Plot?

Now, let's imagine you surveyed **multiple days**, and you want to track how the time spent on each activity changes over a week.

A **Stack Plot** is a type of area chart that helps visualize **multiple quantities over time**, stacked on top of each other. It's especially useful to see **how individual parts contribute to a whole over time**.

Use Cases:

- Time spent on different activities over days
- Distribution of tasks by team members over a project timeline
- Website traffic sources over a week

Stack Plot Example

```python
import matplotlib.pyplot as plt

days = [1, 2, 3, 4, 5, 6, 7]  # Days of the week
studying = [3, 4, 3, 5, 4, 3, 4]
playing = [2, 2, 1, 1, 2, 3, 2]
watching_tv = [2, 1, 2, 2, 1, 1, 1]
sleeping = [5, 5, 6, 5, 6, 5, 5]

labels = ['Studying', 'Playing', 'Watching TV', 'Sleeping']
colors = ['skyblue', 'lightgreen', 'gold', 'lightcoral']

plt.figure(figsize=(10,6))
plt.stackplot(days, studying, playing, watching_tv, sleeping,
              labels=labels, colors=colors, alpha=0.8)

plt.legend(loc='upper left')  # Location of the legend
plt.title('Weekly Activity Tracker')
plt.xlabel('Day')
plt.ylabel('Hours')
plt.grid(True)
plt.show()
```

Key Parameters in `stackplot()`

Parameter	Description
`x`	The x-axis data (like days)
`*args`	Multiple y-values (like studying, playing, etc.)
`labels`	Labels for the legend
`colors`	List of colors for each stack
`alpha`	Transparency level (0 to 1)
`loc` (in `legend`)	Position of the legend (`'upper left'`, `'best'`, etc.)

Summary

- Use **pie charts** for a snapshot in time.
- Use **stack plots** to see how data changes over time while still showing parts of a whole.
- Customize your plot using parameters like `colors`, `labels`, `alpha`, and `legend`.

Stack plots are a great way to **tell a story over time**.

Quick quiz

Try making a stackplot with your own weekly schedule!

Histograms in Matplotlib

A **histogram** is a type of plot that shows the distribution of a dataset. It's especially useful for visualizing the **frequency of numerical data** within specified ranges (called *bins*).

Why and When to Use Histograms?

Use histograms when:

- You want to **understand the distribution** of a numerical dataset (e.g. age, salary, views).
- You want to **detect skewness**, outliers, or **understand spread**.
- You're **binning continuous data** into intervals.

Examples include:

- Analyzing the age of your YouTube viewers

- Understanding test scores distribution
- Checking if data is normally distributed

Understanding the `bins` Argument

The `bins` argument controls how the data is grouped:

- If an **integer**, it defines the number of equal-width bins.
- If a **list**, it defines custom bin **edges**, allowing you to control the range and width of each bin.

```
plt.hist(data, bins=10)  # 10 equal-width bins
plt.hist(data, bins=[10, 20, 30, 40, 60, 100])  # Custom age bins
```

`edgecolor` for Better Visibility

The `edgecolor` parameter adds borders to the bars, improving clarity:

```
plt.hist(data, bins=10, edgecolor='black')
```

Example: Age Distribution of YouTube Viewers

Here is age data for some viewers:

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

ages = [
    34, 28, 36, 45, 27, 27, 45, 37, 25, 35,
    25, 25, 32, 10, 12, 24, 19, 33, 20, 15,
    44, 27, 30, 15, 24, 31, 18, 33, 23, 27,
    23, 48, 29, 19, 38, 17, 32, 10, 16, 31,
    37, 31, 28, 26, 15, 22, 25, 40, 33, 12,
    33, 26, 23, 36, 40, 39, 21, 26, 33, 39,
    25, 28, 18, 18, 38, 43, 29, 40, 33, 23,
    33, 45, 29, 45, 3, 38, 30, 27, 30, 10,
    27, 33, 44, 24, 21, 24, 39, 33, 24, 35,
    30, 39, 22, 26, 26, 15, 32, 32, 30, 27
]

bins = [10, 20, 30, 40, 50, 60, 70]

plt.hist(ages, bins=bins, edgecolor='black')
plt.title('Age Distribution of YouTube Viewers')
plt.xlabel('Age Group')
plt.ylabel('Number of Viewers')
plt.grid(True, linestyle='--', alpha=0.5)
plt.show()
```

Adding a Vertical Line: `axvline`

Use `axvline` to mark a specific value like the **average age** or a **threshold**.

```
plt.hist(ages, bins=bins, edgecolor='black')
plt.axvline(np.mean(ages), color='red', linestyle='--', linewidth=2,
label='Average Age')
plt.legend()
plt.title('Age Distribution with Mean Line')
plt.show()
```

Summary

Parameter	Purpose
`bins`	Number or custom edges of bins
`edgecolor`	Color around each bar
`axvline`	Vertical reference line

Scatter Plot in Matplotlib

Scatter plots are used to show the relationship between two variables. Let's assume we are plotting the **study hours vs. exam scores** for a group of students.

1. Basic Scatter Plot

```
import matplotlib.pyplot as plt

# Sample data
study_hours = [1, 2, 3, 4, 5, 6, 7, 8, 9]
exam_scores = [40, 45, 50, 55, 60, 65, 75, 85, 90]

plt.scatter(study_hours, exam_scores)
plt.title('Study Hours vs Exam Score')
plt.xlabel('Study Hours')
plt.ylabel('Exam Score')
plt.grid(True)
plt.show()
```

> This shows a simple scatter plot. You can notice a general upward trend — more study hours lead to better scores.

---

2. Adding Color and Size

```
# Size of points based on score (bigger score -> bigger point)
sizes = [score * 2 for score in exam_scores]
colors = ['red' if score < 60 else 'green' for score in exam_scores]

plt.scatter(study_hours, exam_scores, s=sizes, c=colors)
plt.title('Colored & Sized Scatter Plot')
plt.xlabel('Study Hours')
plt.ylabel('Exam Score')
plt.grid(True)
plt.show()
```

> Points are colored based on performance and scaled in size by score. Red means low score, green means good.

---

3. Using a Colormap

```
import numpy as np

# Also works with Numpy Arrays
scores_normalized = np.array(exam_scores)

plt.scatter(study_hours, exam_scores, c=scores_normalized, cmap='viridis')
plt.colorbar(label='Score')
plt.title('Scatter Plot with Colormap')
plt.xlabel('Study Hours')
plt.ylabel('Exam Score')
plt.grid(True)
plt.show()
```

> Google: https://matplotlib.org/stable/users/explain/colors/colormaps.html
>
> `cmap` adds gradient coloring based on the score. `colorbar` helps understand what the colors represent.

---

4. Adding Annotations

```
plt.scatter(study_hours, exam_scores)

# Add labels
```

```
for i in range(len(study_hours)):
    plt.annotate(f'Student {i+1}', (study_hours[i], exam_scores[i]))

plt.title('Scatter Plot with Annotations')
plt.xlabel('Study Hours')
plt.ylabel('Exam Score')
plt.grid(True)
plt.show()
```

> Annotations help identify individual points, which is useful in small datasets.

---

5. Multiple Groups in One Plot

```
# Assume two groups: Class A and Class B
class_a_hours = [2, 4, 6, 8]
class_a_scores = [45, 55, 65, 85]

class_b_hours = [1, 3, 5, 7, 9]
class_b_scores = [40, 50, 60, 70, 90]

plt.scatter(class_a_hours, class_a_scores, label='Class A', color='blue')
plt.scatter(class_b_hours, class_b_scores, label='Class B', color='orange')

plt.title('Scatter Plot: Class A vs Class B')
plt.xlabel('Study Hours')
plt.ylabel('Exam Score')
plt.legend()
plt.grid(True)
plt.show()
```

> When comparing two datasets, use different colors and a legend for clarity.

Subplots in Matplotlib

Subplots allow you to show **multiple plots in a single figure**, side-by-side or in a grid layout. This is helpful when comparing different datasets or aspects of the same data.

---

1. Basic Subplot (1 row, 2 columns)

```
import matplotlib.pyplot as plt

# Data
x = [1, 2, 3, 4, 5]
y1 = [i * 2 for i in x]
y2 = [i ** 2 for i in x]
```

```
# Create a figure with 1 row and 2 columns
plt.subplot(1, 2, 1)  # (rows, cols, plot_no)
plt.plot(x, y1)
plt.title('Double of x')

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

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

> `plt.subplot(1, 2, 1)` means 1 row, 2 columns, and we're plotting in the 1st subplot.

---

2. 2×2 Grid of Subplots

```
# More variations of x
y3 = [i ** 0.5 for i in x]
y4 = [10 - i for i in x]

plt.figure(figsize=(8, 6))  # Optional: make it bigger

plt.subplot(2, 2, 1)
plt.plot(x, y1)
plt.title('x * 2')

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

plt.subplot(2, 2, 3)
plt.plot(x, y3)
plt.title('sqrt(x)')

plt.subplot(2, 2, 4)
plt.plot(x, y4)
plt.title('10 - x')

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

> This lays out 4 plots in a 2x2 grid. `plt.tight_layout()` avoids overlapping titles and labels.

---

3. Using `plt.subplots()` for Clean Code

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

axs[0].plot(x, y1)
axs[0].set_title('x * 2')

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

fig.suptitle('Simple Comparison Plots', fontsize=14)
fig.tight_layout()
fig.subplots_adjust(top=0.85)  # So title doesn't overlap
fig.savefig('my_plots.png')    # Save as image

plt.show()
```

So to summarize:

- axs is for working on individual plots
- fig is for settings that apply to the whole figure

> `plt.subplots()` returns a figure and a list/array of axes objects. This is more flexible and cleaner, especially for loops or advanced customizations.

---

4. Looping Over Subplots

```
fig, axs = plt.subplots(2, 2, figsize=(8, 6))
ys = [y1, y2, y3, y4]
titles = ['x * 2', 'x squared', 'sqrt(x)', '10 - x']

for i in range(2):
    for j in range(2):
        idx = i * 2 + j
        axs[i, j].plot(x, ys[idx])
        axs[i, j].set_title(titles[idx])

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

> This approach works well when dealing with dynamic or repetitive data series.

Introduction to Seaborn

Seaborn is a Python library built on top of Matplotlib that makes it **easier** and **prettier** to create complex, beautiful visualizations.

Why Seaborn?

- Matplotlib is powerful but very **low-level**.
- Seaborn adds **high-level** features like **automatic styling, themes, color palettes**, and **dataframe integration**.
- Seaborn comes with built in Datasets
- Makes complex plots (like **boxplots**, **violin plots**, **heatmaps**, **pairplots**) **very easy**.

In short:

- Less code
- Better-looking graphs
- Easy handling of DataFrames (like from pandas)

Basic Setup

```
# Install Seaborn if you don't have it
!pip install seaborn

# Import Seaborn
import seaborn as sns
import matplotlib.pyplot as plt
```

Seaborn Themes

Seaborn automatically makes your plots look good, but you can even control the overall "theme."

```
sns.set_theme(style="darkgrid")  # Options: whitegrid, dark, white, ticks
```

Example:

```
import numpy as np

x = np.array([0, 2, 3, 4, 5, 6, 7, 8, 9 ,10, 60])
y = np.sin(x)

sns.lineplot(x=x, y=y)
plt.title('Beautiful Line Plot')
plt.show()
```

Summary

- Seaborn makes complex, attractive, and statistical plots simple and ready for professional reports.

- Matplotlib is important to know for fine-tuning or customization, but Seaborn should be your first choice for day-to-day plotting.
- In real-world Data Science projects, Seaborn saves hours of manual work by offering higher-level, smarter defaults.

Basic Plot Types in Seaborn

- **Seaborn** comes with **built-in example datasets**.
- These are small real-world datasets like restaurant tips, flight passenger counts, iris flower measurements, etc.
- They are mainly used for **practice**, **examples**, and **learning** plotting techniques without needing to manually download any data.

You can **load** these datasets directly into a **pandas DataFrame** using `sns.load_dataset()`.

How to See All Available Datasets

```
import seaborn as sns

print(sns.get_dataset_names())
```

This will list dataset names like: `tips`, `flights`, `iris`, `diamonds`, `penguins`, `titanic`, etc.

How to Load a Dataset

```
tips = sns.load_dataset('tips')
print(tips.head())
```

- This loads the **"tips"** dataset (restaurant bills and tips).
- It returns a **pandas DataFrame** ready for analysis or plotting.

Why Seaborn Provides Datasets

- To **quickly test** different types of plots
- To **learn plotting** without needing your own data at first
- To create **examples** and **tutorials** easily
- To show real-world messy data handling (missing values, categorical data, etc.)

Now lets look into some plots we can create usign Seaborn

1. Line Plot

```
tips = sns.load_dataset('tips')

sns.lineplot(x="total_bill", y="tip", data=tips)
plt.title('Line Plot Example')
plt.show()
```

---

2. Scatter Plot

```
sns.scatterplot(x="total_bill", y="tip", data=tips, hue="time")
plt.title('Scatter Plot with Color by Time')
plt.show()
```

---

3. Bar Plot

```
sns.barplot(x="day", y="total_bill", data=tips)
plt.title('Average Bill per Day')
plt.show()
```

---

4. Box Plot

Boxplots show distributions, medians, and outliers in one simple plot.

```
sns.boxplot(x="day", y="total_bill", data=tips)
plt.title('Boxplot of Total Bill per Day')
plt.show()
```

---

5. Heatmap (Correlation Matrix)

```
flights = sns.load_dataset('flights')
pivot_table = flights.pivot("month", "year", "passengers")

sns.heatmap(pivot_table, annot=True, fmt="d", cmap="YlGnBu")
plt.title('Heatmap of Passengers')
plt.show()
```

Working with Pandas DataFrames

One of the biggest strengths of Seaborn:

```
import pandas as pd

df = pd.DataFrame({
    "age": [22, 25, 47, 52, 46, 56, 55, 60, 34, 43],
    "salary": [25000, 27000, 52000, 60000, 58000, 62000, 61000, 65000, 38000,
45000],
    "gender": ["M", "F", "M", "F", "F", "M", "M", "F", "F", "M"]
})

sns.scatterplot(x="age", y="salary", hue="gender", data=df)
plt.title('Salary vs Age Scatter Plot')
plt.show()
```

Summary

Feature	Matplotlib	Seaborn
Default Styles	Basic	Beautiful
Syntax Level	Low	High
Works with DataFrames	Manual	Easy
Plotting Complex Graphs	Tedious	Very Easy

Final Words

- **Seaborn** makes data visualization **faster, prettier, and smarter**.
- You should still know Matplotlib basics (for fine-tuning plots).
- In real-world Data Science, we usually **start with Seaborn**, and **customize with Matplotlib**.
