Part 5:

Data Science and Machine Learning With Amin.

Data Visualization with Matplotlib

Amin Hydar Ali

Table of Contents

Introduction to Matplotlib

Matplotlib is the foundation of nearly all data visualization in Python.
Think of it as the **artist's canvas** and **paintbrush set** that most other visualization libraries build upon.

Before plotting anything meaningful, we must understand:

- what Matplotlib is
- how it thinks
- why it has two interfaces (Pyplot vs OO)
- what Figures and Axes mean
- how a basic plot works
- and how to properly import the library

This lesson lays the groundwork for everything else.

What is Matplotlib?

Matplotlib is a Python library for creating **2D plots**, such as:

- line charts
- bar charts
- scatter plots
- histograms
- pie charts
- heatmaps
- complex multi-panel scientific figures

Think of Matplotlib like this:

- **Figure** → the entire piece of paper
- **Axes** → a single drawing area (like one box on graph paper)
- **Plots** → the lines, bars, or points you draw inside an Axes

Matplotlib is extremely flexible, from simple quick plots to highly customized scientific visualizations.

Why do we use Matplotlib?

Because it gives you:

- **complete control** over every part of your visualization
- **professional-level graphics** for research, engineering, finance, etc.
- **integration** with Pandas, NumPy, SciPy, Seaborn
- a foundation that other libraries (Seaborn, Pandas plotting, Plotly) build on

Think of it this way:

Matplotlib is the "photoshop" of Python plotting.

You can create simple sketches or incredibly detailed masterpieces, all depending on how deep you go.

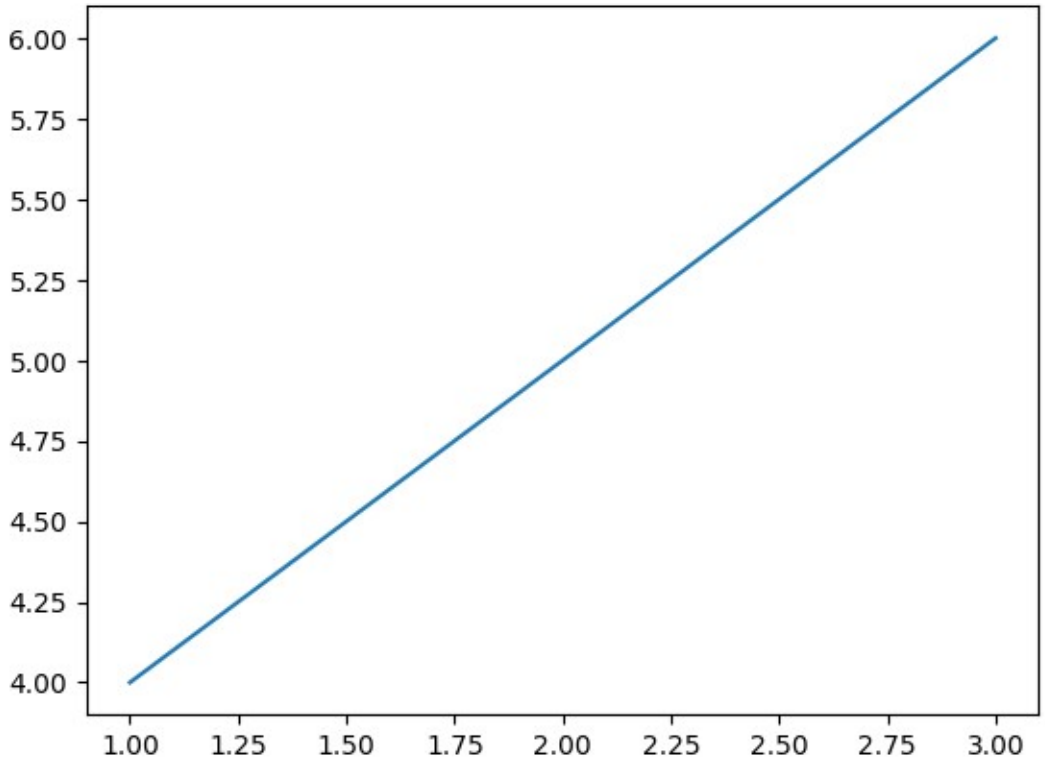
Pyplot vs Object-Oriented (OO) API

Matplotlib has **two different ways** to create plots.
This confuses beginners, so let's break it down carefully.

Pyplot Interface (Quick + Simple)

This is the easiest way to start.
You call functions directly:

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



Pyplot is like using a **whiteboard**:

- You pick up the marker
- Draw on the one board available
- Very quick
- But not very structured

✔ When to use Pyplot:

- quick prototypes
- small scripts
- teaching
- simple plots

Object-Oriented (OO) Interface (Professional + Structured)

Instead of drawing directly, you create **objects** (Figure and Axes) and draw *on them*:

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

OO is like designing a **digital canvas**:

- You explicitly create a large canvas (Figure)
- Then create drawing areas (Axes)
- You tell each Axes exactly what to draw
- Full control over layout

✔ When to use OO:

- dashboards
- multi-panel figures
- complex custom layouts
- professional reports
- scientific publications

WHY Matplotlib has 2 interfaces

When Matplotlib was created, Pyplot was meant to mimic MATLAB's quick plotting style.

Later, users needed **more flexibility**, so the OO API was introduced, but Pyplot was kept for convenience.

Pyplot = quick convenience
OO = professional control

Both do the same thing under the hood, but OO keeps everything explicit and organized.

Understanding Figure & Axes (The Most Important Concept)

Many learners struggle here, so we'll explain it deeply.

Figure (the big canvas)

A Figure is the **entire window or page** where your plots live.

- It can contain **one** plot
- Or **many** subplots
- It controls the overall size, background, DPI, layout

Think of it this way:

> **Figure** = the sheet of paper.

You cannot draw anything without a Figure.

Axes (the actual plot area)

This is where:

- data is plotted
- **x**/**y** axes appear
- labels, ticks, and grids live
- the title is displayed

Think of it this way:

> **Axes** = the graph box you draw inside.

A Figure can have:

- 1 Axes
- 4 Axes (2×2 grid)
- 10 Axes
- Any custom arrangement

Each Axes is independent.

Example:

You create a Figure and a single Axes:

```
1. fig, ax = plt.subplots()
```

Now:

- `fig` is the whole canvas
- `ax` is one plot area
- You draw on `ax`:

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

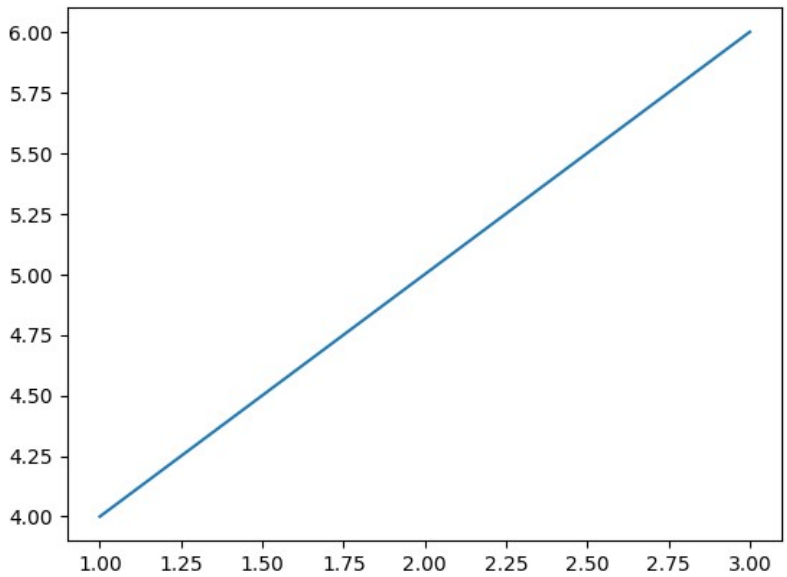
Everything becomes very clear once you see this mental model.

First Basic Plot (Step-by-Step Explanation)

Let's create our first simple line plot.

Using Pyplot

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



What happens step-by-step?

1. **`plt.plot()`** tells Matplotlib to draw a line using your data.
2. Pyplot implicitly creates a **Figure and Axes** for you.
3. **`plt.show()`** displays the window containing the figure.

This is the simple, MATLAB-style method.

Using Object-Oriented Style

```
1. import matplotlib.pyplot as plt
2. fig, ax = plt.subplots()
3.
4. ax.plot([1, 2, 3], [4, 5, 6])
5. plt.show()
```

What happens step-by-step?

1. **`plt.subplots()`** explicitly creates
    - a **Figure** (`fig`)
    - an **Axes** (`ax`)
2. You draw on **`ax`** using **`ax.plot()`**
3. **`plt.show()`** displays the result

This is cleaner, more powerful, and preferred for anything beyond basic work.

Importing Matplotlib Correctly (Why it Matters)

The standard import is:

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

Why this import is used everywhere:

- **`pyplot`** provides the simple interface
- **`plt`** is short, consistent, and widely recognized
- Most tutorials, documentation, and code examples use **`plt`**

This consistency is important for collaboration and readability.

Import conventions matter.

Just like Pandas uses **`pd`**, Matplotlib uses **`plt`**.

Practical Exercise: Create Your First Line Plot Using BOTH Styles

Pyplot Version

```
import matplotlib.pyplot as plt

plt.plot([0, 1, 2, 3], [10, 20, 25, 30])
plt.title("My First Pyplot Line Plot")
plt.xlabel("X Values")
plt.ylabel("Y Values")

plt.show()
```

OO Version

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

ax.plot([0, 1, 2, 3], [10, 20, 25, 30])
ax.set_title("My First OO Line Plot")
ax.set_xlabel("X Values")
ax.set_ylabel("Y Values")

plt.show()
```

Notice the difference:

Feature	Pyplot	OO Style
Creates Figure/Axes automatically?	✔ Yes	✖ No (you create them)
Best for	quick plots	complex & professional work
Control level	low	high
Clarity	less explicit	very explicit

Final Summary of Lesson 1

Concept	Key Idea
What is Matplotlib?	A visualization library that acts like your drawing canvas in Python
Pyplot API	Quick, simple, MATLAB-style plotting
OO API	Professional, explicit, fully controlled plotting
Figure	Entire drawing canvas
Axes	The actual area where the plot is drawn
First Plot	Built using either Pyplot or OO methods
Import	Always use `import matplotlib.pyplot as plt`

Lesson 2: Basic Plot Types

Data visualization is not about "making things look nice."
It is about choosing the correct visual language for the question you are asking.

Each plot type exists for a reason.
Using the wrong plot is like using the wrong word in a sentence, the message becomes unclear or misleading.

In this lesson, we'll break down why each plot type exists, what it communicates, and how to create it properly in Matplotlib.

1. Line Plots - Showing Change Over Time

What a Line Plot Is Really Doing

A line plot shows how one variable changes in relation to another, usually **time**.

Think of a line plot as:

> **Connecting the dots to tell a story of movement.**
>
> Each point represents a moment.
> The line shows *continuity* between moments.

When to Use a Line Plot

Use line plots when:

- order matters
- data is continuous
- you want to see trends
- you care about direction (up/down)

Real-world examples:

- daily sales
- stock prices
- temperature over time

- website traffic

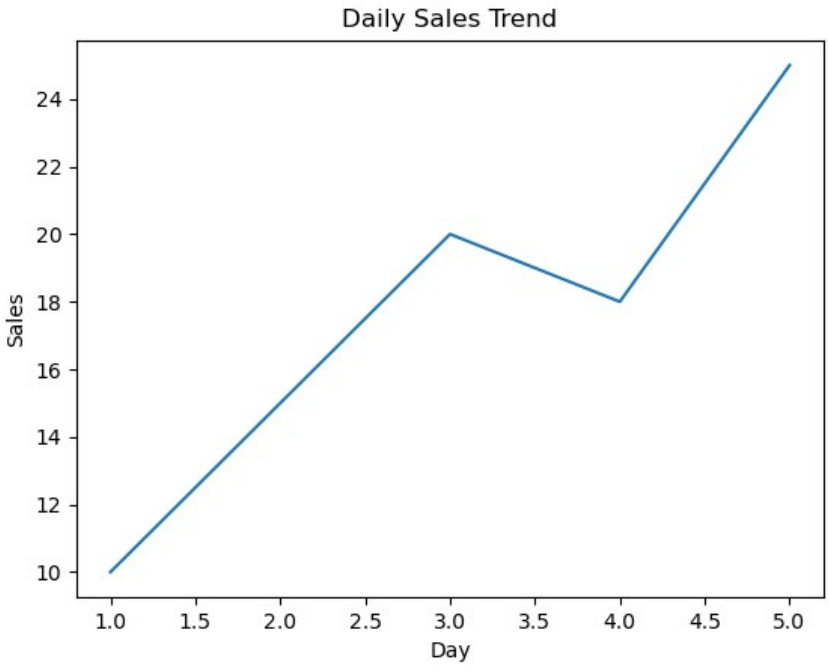
How Matplotlib Draws a Line Plot

Matplotlib:

1. Places points at `(x, y)`
2. Connects them **in the order they appear**
3. Assumes continuity between points

Example: Line Plot

```
1. import matplotlib.pyplot as plt
2.
3. x = [1, 2, 3, 4, 5]
4. y = [10, 15, 20, 18, 25]
5.
6. plt.plot(x, y)
7. plt.title("Daily Sales Trend")
8. plt.xlabel("Day")
9. plt.ylabel("Sales")
10. plt.show()
```



Why this works:

- The x-axis has order
- The line emphasizes *trend* rather than individual points

2. Scatter Plots - Showing Relationships (Not Order)

What a Scatter Plot Is Really Saying

A scatter plot asks:

"Do these two variables relate to each other?"

Unlike line plots:

- order does **not** matter
- points are **not connected**

Each dot stands alone.

When to Use a Scatter Plot

Use scatter plots when:

- you want to see correlation
- you want to detect patterns or clusters
- you want to find outliers

Real-world examples:

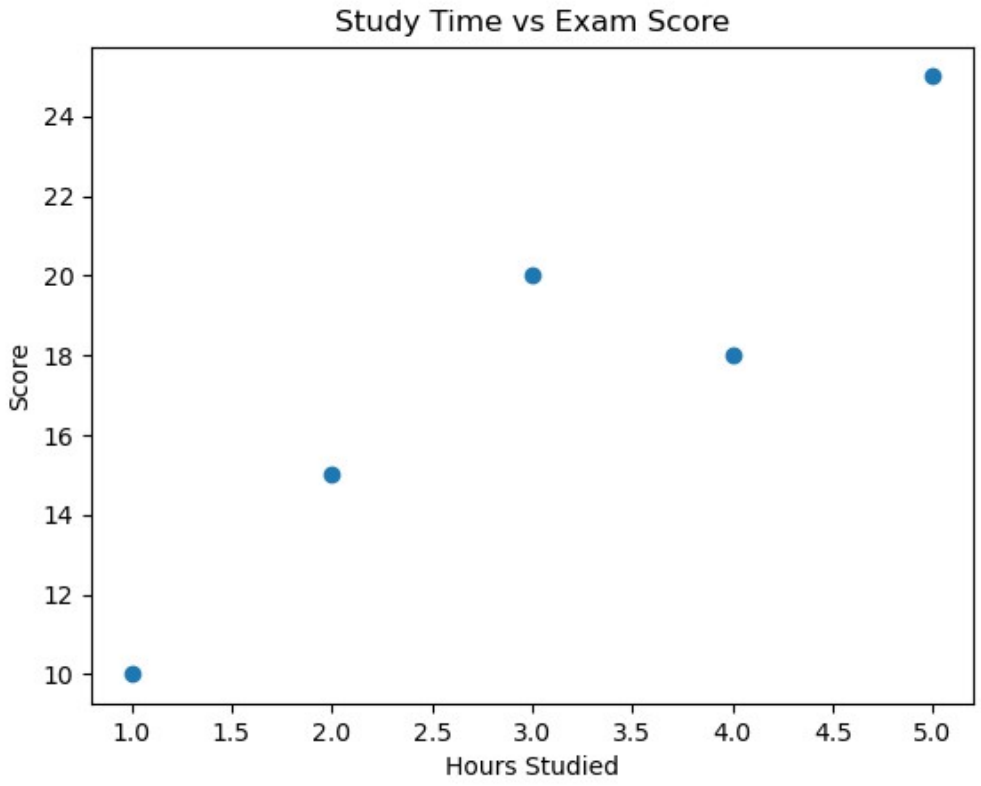
- height vs weight
- study time vs exam score
- advertising spend vs revenue

How Matplotlib Draws a Scatter Plot

- Each point is drawn independently
- No implied relationship between neighboring points
- Position alone carries meaning

Example: Scatter Plot

```
1. plt.scatter(x, y)
2. plt.title("Study Time vs Exam Score")
3. plt.xlabel("Hours Studied")
4. plt.ylabel("Score")
5. plt.show()
```



Key idea:

A scatter plot does NOT imply progression, only association.

3. Bar Charts - Comparing Categories

Bar charts compare discrete categories, not continuous values.

What Bar Charts Communicate

A bar chart answers:

“How much does each category have?”

Bar length encodes magnitude.

When to Use Bar Charts

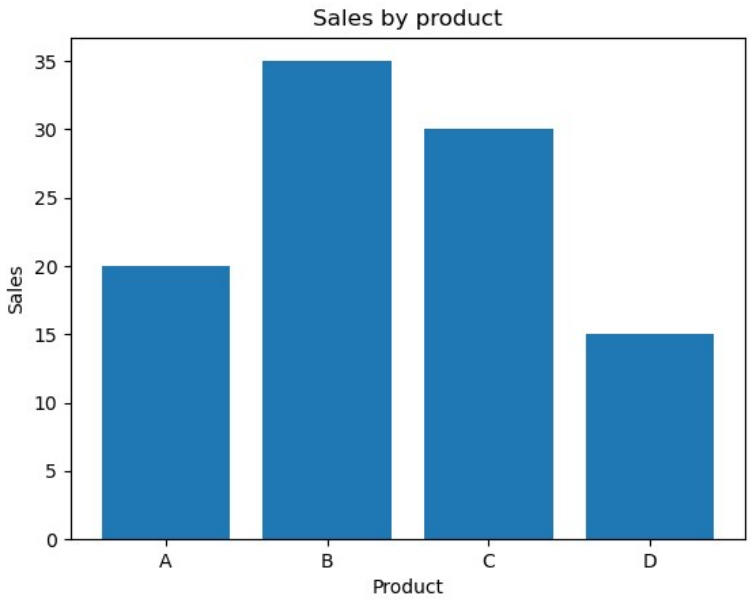
Use bar charts when:

- values belong to categories
- comparison is the goal
- categories are distinct

Real-world examples:

- sales per product
- students per class
- votes per candidate

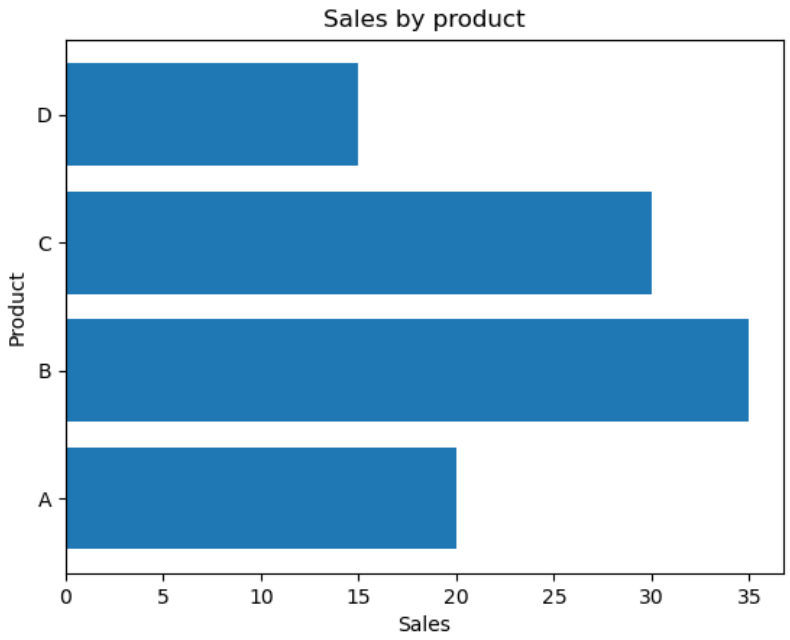
Vertical Bar Chart




```
categories = ["A", "B", "C", "D"]
values = [20, 35, 30, 15]

plt.bar(categories, values)
plt.title("Sales by Product")
plt.xlabel("Product")
```

```
7. plt.ylabel("Sales")
8. plt.show()
```

Horizontal Bar Chart




```
1. plt.barh(categories, values)
2. plt.title("Sales by Product")
3. plt.xlabel("Sales")
4. plt.ylabel("Product")
5. plt.show()
```

Why horizontal bars exist:

- long category names
- easier ranking from top to bottom

4. Histograms - Understanding Distributions

What a Histogram Really Shows

A histogram answers:

“How is my data distributed?”

It shows:

- frequency
- spread
- skewness
- clusters
- gaps

When to Use Histograms

Use histograms when:

- data is numeric and continuous
- you want to see shape, not exact values

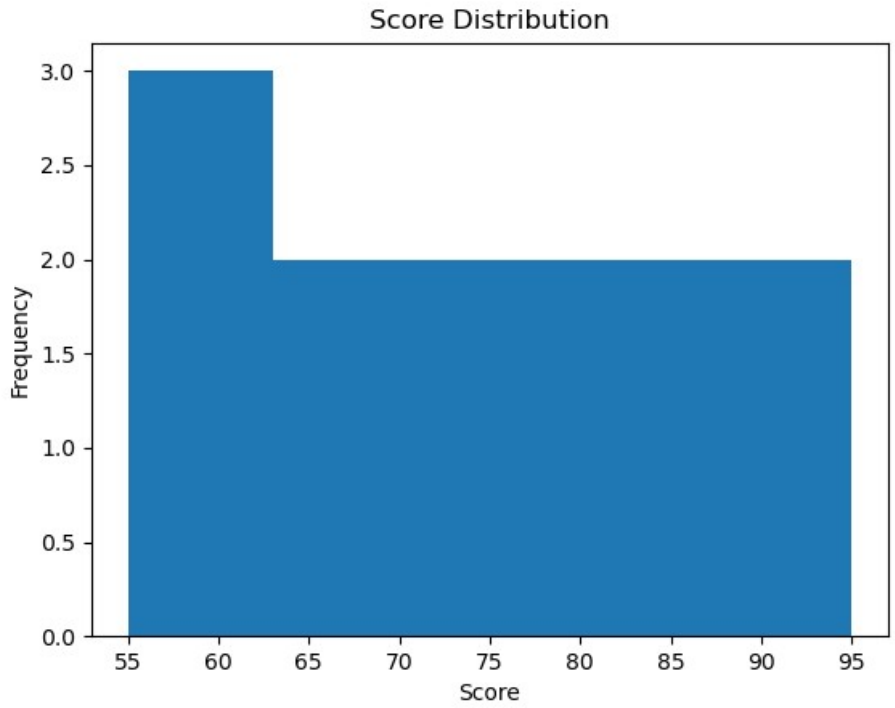
Real-world examples:

- exam scores
- ages
- income distribution

How Matplotlib Builds a Histogram

1. Splits data into **bins**
2. Counts how many values fall into each bin
3. Draws bars representing frequency

Example: Histogram




```
1. scores = [55, 60, 62, 65, 70, 72, 75, 80, 85, 90, 95]
2.
3. plt.hist(scores, bins=5)
4. plt.title("Score Distribution")
5. plt.xlabel("Score")
6. plt.ylabel("Frequency")
7. plt.show()
```

Why bins matter:

- too few ⟶ oversimplified
- too many ⟶ noisy

5. Pie Charts - Showing Proportions (Use Carefully)

What a Pie Chart Shows

A pie chart shows **parts of a whole**.

Total = 100%

Each slice is a percentage.

When to Use Pie Charts

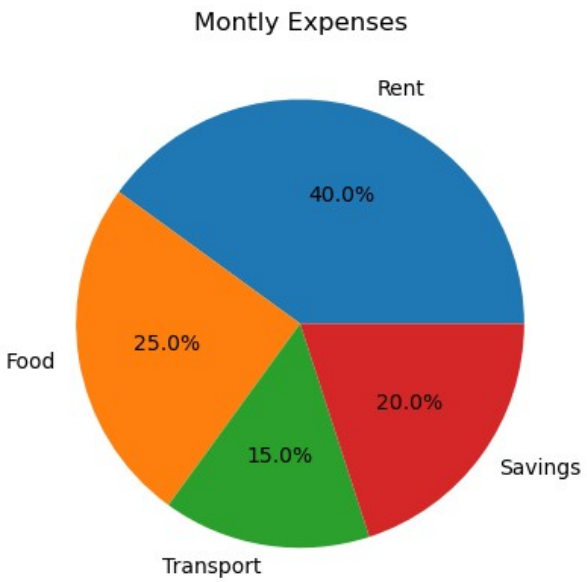
Use pie charts when:

- there are few categories
- proportions matter
- values add up to a whole

Avoid pie charts when:

- too many categories
- precise comparisons matter

Example: Pie Chart




```
labels = ["Rent", "Food", "Transport", "Savings"]
sizes = [40, 25, 15, 20]

plt.pie(sizes, labels=labels, autopct="%1.1f%%")
plt.title("Monthly Expenses")
plt.show()
```

Why `autopct` matters:

- humans struggle to estimate angles
- percentages improve readability

Practical: Plot Sample Data Using Each Chart Type

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [10, 15, 20, 18, 25]

categories = ["A", "B", "C"]
values = [30, 50, 20]

data = [5, 7, 8, 9, 10, 12, 15, 18, 20]

fig, axes = plt.subplots(2, 3, figsize=(15, 8))

axes[0, 0].plot(x, y)
axes[0, 0].set_title("Line Plot")

axes[0, 1].scatter(x, y)
axes[0, 1].set_title("Scatter Plot")

axes[0, 2].bar(categories, values)
axes[0, 2].set_title("Bar Chart")

axes[1, 0].barh(categories, values)
axes[1, 0].set_title("Horizontal Bar")

axes[1, 1].hist(data, bins=5)
axes[1, 1].set_title("Histogram")

axes[1, 2].pie(values, labels=categories, autopct="%1.1f%%")
axes[1, 2].set_title("Pie Chart")

plt.tight_layout()
plt.show()

```

Key Takeaways

Plot Type	Best For
Line	Trends & change over time
Scatter	Relationships & correlation
Bar	Comparing categories
Histogram	Distribution & spread
Pie	Proportions of a whole

Lesson 3: Customizing Plots

Creating a plot is only half the job.
A plot that looks default, unlabeled, or cluttered may technically be "correct", but it often fails to communicate.

Customization is about:

- clarity
- storytelling
- guiding the viewer's eyes
- professional credibility

Think of this lesson as learning visual communication, not decoration.

1. Titles, Axis Labels, and Legends

Why These Matter

A plot without labels is like a book without chapter titles.

The viewer should never have to guess:

- what the plot shows
- what each axis means
- what each line or color represents

Plot Title

What it does:

- explains the overall message
- sets context immediately

```
1. ax.set_title("Monthly Sales Trend")
```

Think of it this way:

The title is the headline of your chart.

A good title answers:
What is changing? Over what?

Axis Labels

Why axis labels matter:

Numbers without meaning are useless.

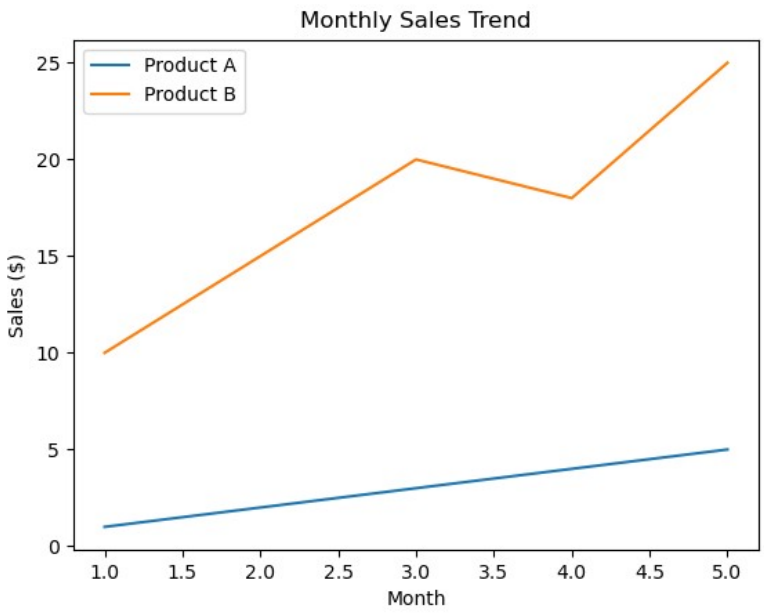
```
1. ax.set_xlabel("Month")
2. ax.set_ylabel("Sales ($)")
```

Think of it this way:

Axis labels are the **units on a measuring tape**.

Legends

Legends explain *which visual element means what.*




```
1. ax.plot(x, y1, label="Product A")
2. ax.plot(x, y2, label="Product B")
3. ax.legend()
```

Why legends matter:

Without them, multiple lines become visual noise.

2. Changing Colors and Markers

Why Color Is Not Decoration

Color:

- separates data
- draws attention
- implies importance

But bad color choices **confuse** instead of clarify.

Changing Line Colors

```
1. ax.plot(x, y, color="green")
```

You can also use:

- named colors (`"red"`, `"blue"`)
- hex codes (`"#1f77b4"`)

Markers (Emphasizing Data Points)

Markers show **individual observations**, not just trends.

```
1. ax.plot(x, y, marker="o")
```

Common markers:

- `"o"` → circle
- `"s"` → square
- `"^"` → triangle

Think of it this way:

Markers are pins on a map, they show exact locations.

3. Line Styles (Communicating Meaning)

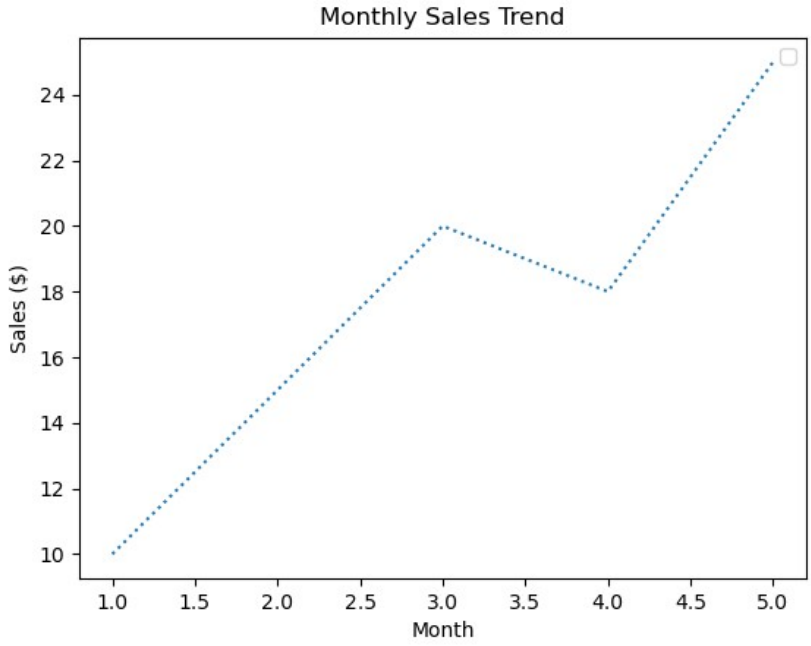
Why Line Style Matters

Line styles allow you to distinguish:

- forecasts vs actuals

- different scenarios
- categories without using color

Common Line Styles




```
1. ax.plot(x, y, linestyle="--")   # dashed
2. ax.plot(x, y, linestyle=":")    # dotted
```

Styles:

- " - " solid
- " - - " dashed
- " : " dotted
- " - . " dash-dot

Think of it this way:

Line styles are accents in speech, subtle but meaningful.

4. Adding Grids (Visual Guidance)

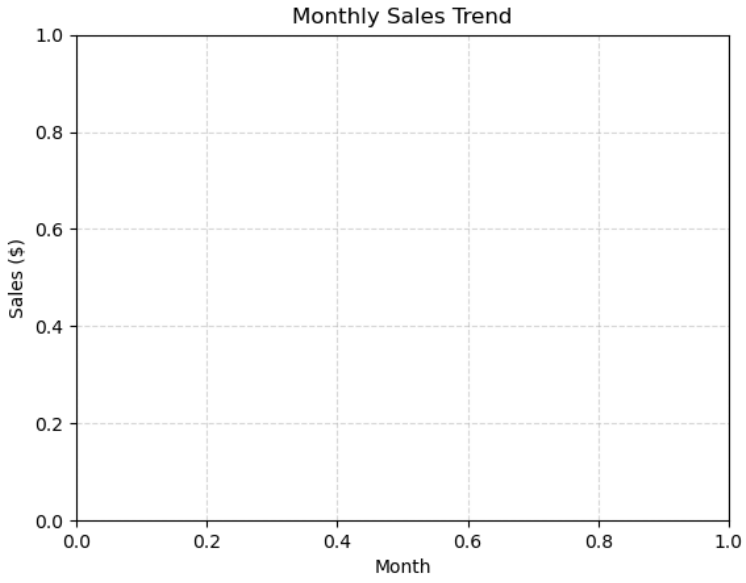
Why Grids Help

Grids:

- improve readability
- help align values
- reduce mental effort

But too many gridlines can clutter.

Adding a Grid




```
1. ax.grid(True)
```

Or customize:

```
1. ax.grid(True, linestyle="--", alpha=0.5)
```

Think of it this way:

Grids are graph paper behind your drawing.

5. Setting X/Y Limits (Controlling Focus)

Why Limits Matter

Auto-scaling is not always ideal.

Sometimes you want to:

- zoom into a region
- avoid misleading scales
- compare charts consistently

Set Axis Limits

```
1. ax.set_xlim(0, 10)
2. ax.set_ylim(0, 100)
```

Warning:

Poor axis limits can distort interpretation.

6. Using Built-in Styles (`plt.style.use`)

What Styles Do

Styles apply global themes:

- colors
- fonts
- grid appearance
- background

Apply a Style

```
1. plt.style.use("ggplot")
```

Popular styles:

- `"default"`
- `"ggplot"`
- `"seaborn-v0_8"`

- `"bmh"`

Think of it this way:

Styles are pre-made design templates.

They ensure visual consistency without manual tweaking.

Practical: Create a Professional-Looking Chart

Below is a complete, polished example using **OO style**.

Goal:

Create a clean, readable, professional time-series chart.

Code Example

```
import matplotlib.pyplot as plt

plt.style.use("seaborn-v0_8")

months = ["Jan", "Feb", "Mar", "Apr", "May"]
sales = [120, 150, 170, 160, 200]

fig, ax = plt.subplots(figsize=(8, 5))

ax.plot(
    months,
    sales,
    color="steelblue",
    marker="o",
    linestyle="-",
    linewidth=2,
    label="Monthly Sales"
)

ax.set_title("Monthly Sales Performance", fontsize=14)
ax.set_xlabel("Month")
ax.set_ylabel("Sales ($)")

ax.grid(True, linestyle="--", alpha=0.6)
ax.legend()

```

```
ax.set_ylim(100, 220)

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

Why This Looks Professional

Feature	Purpose
Style	Visual consistency
Title	Clear context
Labels	Units & meaning
Marker	Data clarity
Grid	Readability
Limits	Focus

Key Takeaways

- Customization improves communication, not aesthetics
- Titles and labels are mandatory, not optional
- Color and style convey meaning
- Grids and limits guide interpretation
- Styles save time and enforce consistency

Lesson 4: Working With Subplots

Most real-world visualizations are not a single chart.
They are collections of related charts that work together to tell a bigger story.

Subplots allow you to:

- compare multiple views of the same data
- show different metrics side by side
- build dashboards and reports
- reduce cognitive load by grouping visuals

Big Idea: What Are Subplots?

A subplot is simply one Axes inside a Figure.

Think of it like this:

Figure = a page in a report
Subplots = panels on that page

A single figure can hold:

- 1 subplot
- 4 subplots
- 20 subplots
  arranged however you want.

1. Creating Multiple Plots in One Figure

Why Multiple Plots Matter

Imagine trying to explain:

- sales trend
- distribution
- category comparison

...using **three separate images**.

Subplots let the viewer see everything **at once**, making comparisons effortless.

Mental Model

A grid layout:

```
1. [ Axes ][ Axes ]
2. [ Axes ][ Axes ]
```

Each Axes:

- has its own data
- its own title
- its own scales

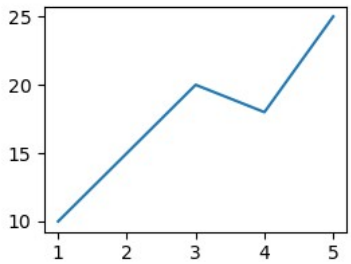
But they all live on the same canvas.

2. `plt.subplot()` vs `plt.subplots()`

This distinction is critical.

`plt.subplot()` (Older, Manual, Less Clear)

```
1. plt.subplot(2, 2, 1)
2. plt.plot(x, y)
```



How it works:

- You specify rows, columns, and position

- Matplotlib creates the Axes implicitly
- You switch "active" Axes behind the scenes

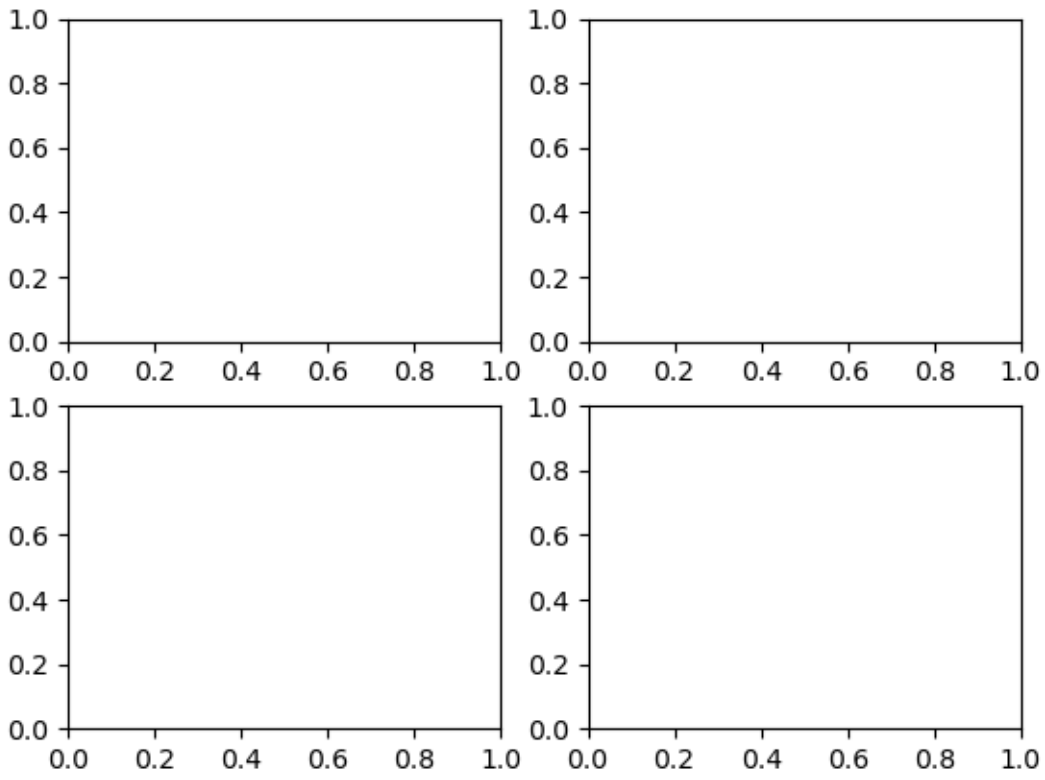
Think of it this way:

`plt.subplot()` is like telling someone where to draw without seeing the paper.

It works, but it's harder to manage.

`plt.subplots()` (Modern, Explicit, Recommended)

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



This creates:

- one Figure
- a grid of Axes objects

You now control **each Axes directly**.

Think of it this way:

> **`plt.subplots()`** is like laying out the paper first, then choosing where to draw.

This is the **professional standard**.

Why `plt.subplots()` Is Better

Feature	subplot	subplots
Explicit Axes	✗	✓
Readability	low	high
Scales to complexity	✗	✓
Industry standard	✗	✓

3. Sharing X/Y Axes

Why Share Axes?

When comparing plots:

- different scales can mislead
- shared axes create honest comparison

Share X or Y Axis

```
1. fig, axes = plt.subplots(2, 2, sharex=True, sharey=True)
```

What this does:

- same x-axis scale for all plots
- same y-axis scale for all plots

Think of it this way:

> Shared axes are like **using the same ruler** for every measurement.

4. Layout Adjustments (`tight_layout`, `figsize`)

Why Layout Matters

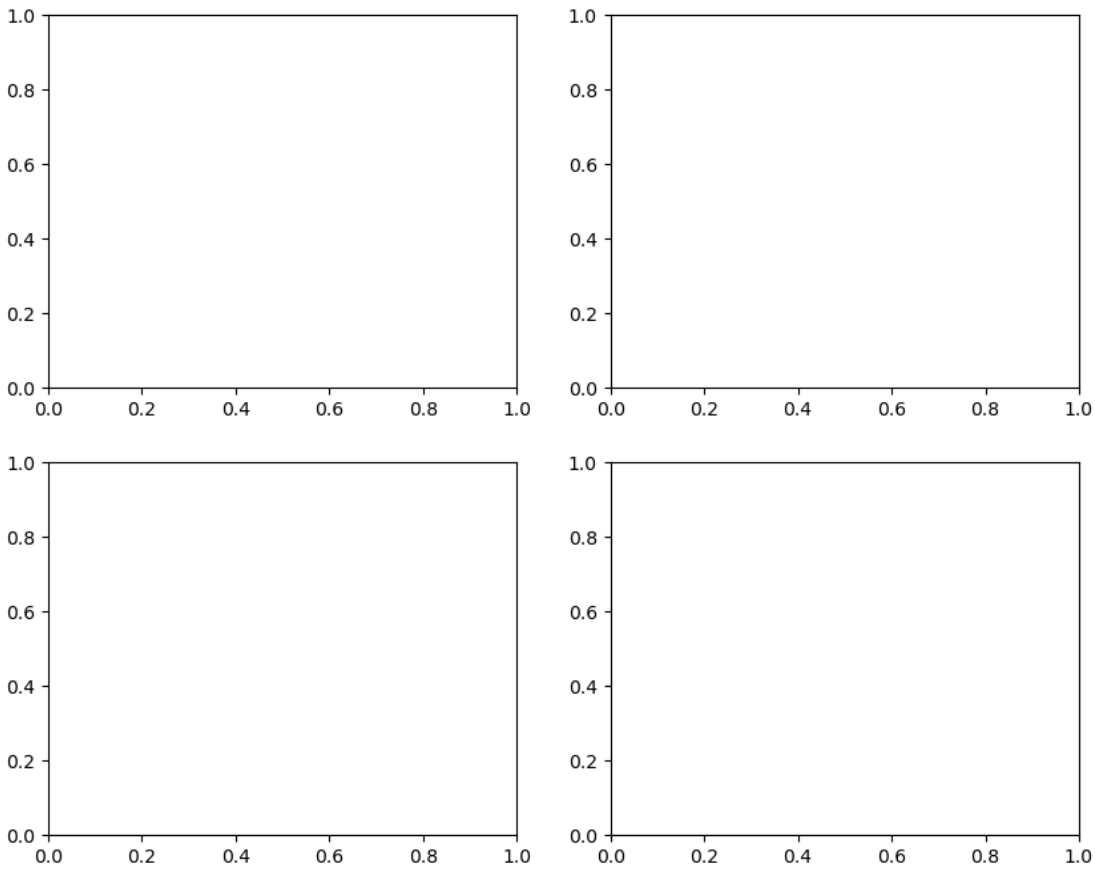
Without adjustments:

- titles overlap
- labels collide
- plots look cramped

This reduces readability.

Figure Size

```
1. fig, axes = plt.subplots(2, 2, figsize=(10, 8))
```



This controls:

- width
- height
- aspect ratio

Think in inches, important for reports and PDFs.

`tight_layout()`

1. plt.tight_layout()

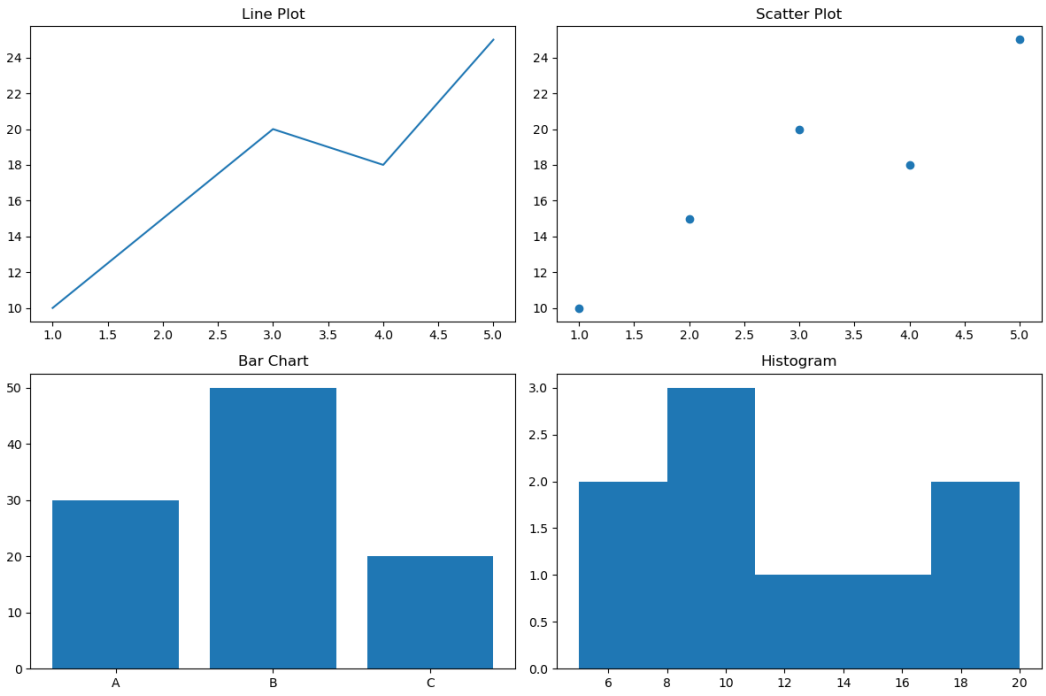
Matplotlib automatically:

- adjusts spacing
- prevents overlaps

Think of it this way:

`tight_layout()` is like auto-arranging furniture in a room.

Practical: Create a 2×2 Grid of Different Chart Types



```python
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [10, 15, 20, 18, 25]
categories = ["A", "B", "C"]
values = [30, 50, 20]
data = [5, 7, 8, 9, 10, 12, 15, 18, 20]

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

axes[0, 0].plot(x, y)
axes[0, 0].set_title("Line Plot")

axes[0, 1].scatter(x, y)
axes[0, 1].set_title("Scatter Plot")

axes[1, 0].bar(categories, values)
axes[1, 0].set_title("Bar Chart")

axes[1, 1].hist(data, bins=5)
axes[1, 1].set_title("Histogram")

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

Lesson 4 Takeaways

- Subplots enable comparison
- **`plt.subplots()`** is the preferred method
- Shared axes improve honesty
- Layout control is essential for readability

Lesson 5: Annotations & Text

Annotations turn a chart from:

> "Here is some data"
> into
> "Here is the insight you should notice."

This is where **storytelling** happens.

1. Why Annotations Matter

Humans do not naturally spot:

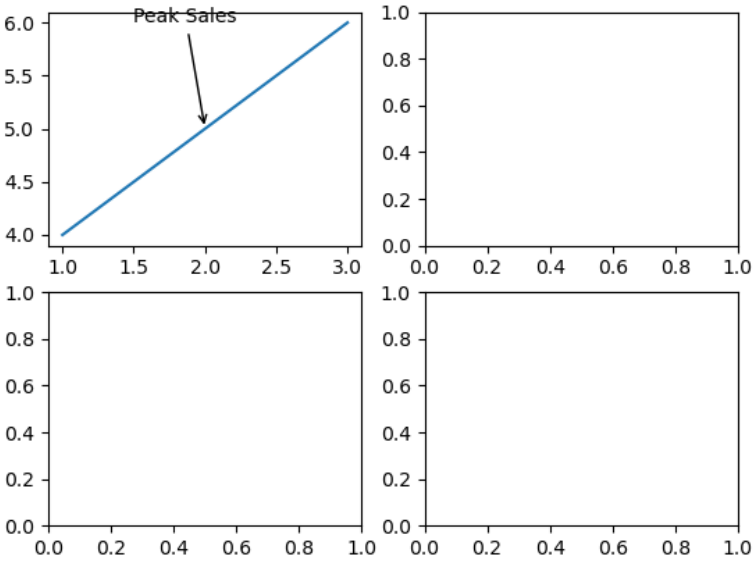
- peaks
- drops
- anomalies
- turning points

Annotations guide attention.

Think of it this way:

> Annotations are highlighters on a textbook page.

2. Annotating Points




```
1. fig,ax = plt.subplots(2,2)
2. ax[0,0].plot([1, 2, 3], [4, 5, 6])
3. ax[0, 0].annotate(
4.     "Peak Sales",
5.     xy=(4, 25),
6.     xytext=(3, 30),
7.     arrowprops=dict(arrowstyle="->")
8. )

9. plt.show()
```

Key parts:

- **`xy`** → point being referenced
- **`xytext`** → text location
- **`arrowprops`** → visual link

3. Adding Arrows

Arrows visually connect:

- text
- data

Without arrows, viewers guess what the label refers to.

4. Custom Text Placement

```
1. ax.text(2, 15, "Promotion Period")
```

Used when:

- no arrow needed
- label is self-explanatory

Think of it this way:

> Text is a sticky note on the chart.

5. Highlighting Areas (Shading)

Used to:

- show time ranges
- emphasize periods
- highlight thresholds

Vertical Shading (Time Ranges)

```
1. ax.axvspan(2, 4, alpha=0.2, color="gray")
```

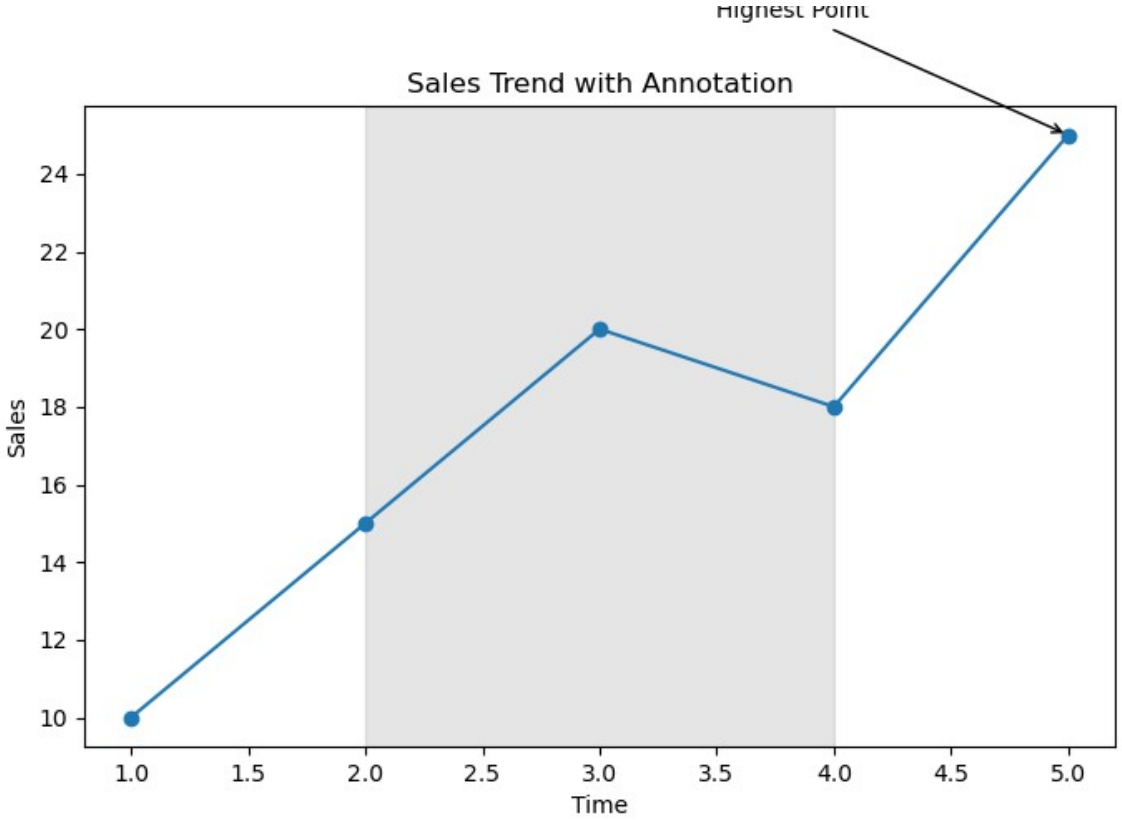
Horizontal Shading (Thresholds)

```
1. ax.axhspan(20, 30, alpha=0.2, color="green")
```

Think of it this way:

> Shading is a spotlight behind the data.

Practical: Annotate Key Points on a Line Chart




```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [10, 15, 20, 18, 25]

fig, ax = plt.subplots(figsize=(8, 5))

ax.plot(x, y, marker="o")

ax.annotate(
    "Highest Point",
    xy=(5, 25),
    xytext=(3.5, 28),
    arrowprops=dict(arrowstyle="->")
)

ax.axvspan(2, 4, alpha=0.2, color="gray")

ax.set_title("Sales Trend with Annotation")
```

```
ax.set_xlabel("Time")
ax.set_ylabel("Sales")

plt.show()
```

Lesson 5 Takeaways

- Annotations guide interpretation
- Arrows connect insight to data
- Text clarifies meaning
- Shading highlights important regions
- Great charts explain themselves

Lesson 6: Time Series Visualization

Time series data is everywhere:

- sales over time
- stock prices
- temperature readings
- website traffic
- sensor data

Visualizing time series is not just plotting dates, it's about handling time correctly, so the chart tells the *right story*.

Big Idea: Why Time Series Is Special

Time is:

- ordered
- continuous
- directional

This means:

- spacing matters
- formatting matters
- comparisons across time must be honest

A bad time-series chart can be actively misleading.

1. Using Pandas Datetime with Matplotlib

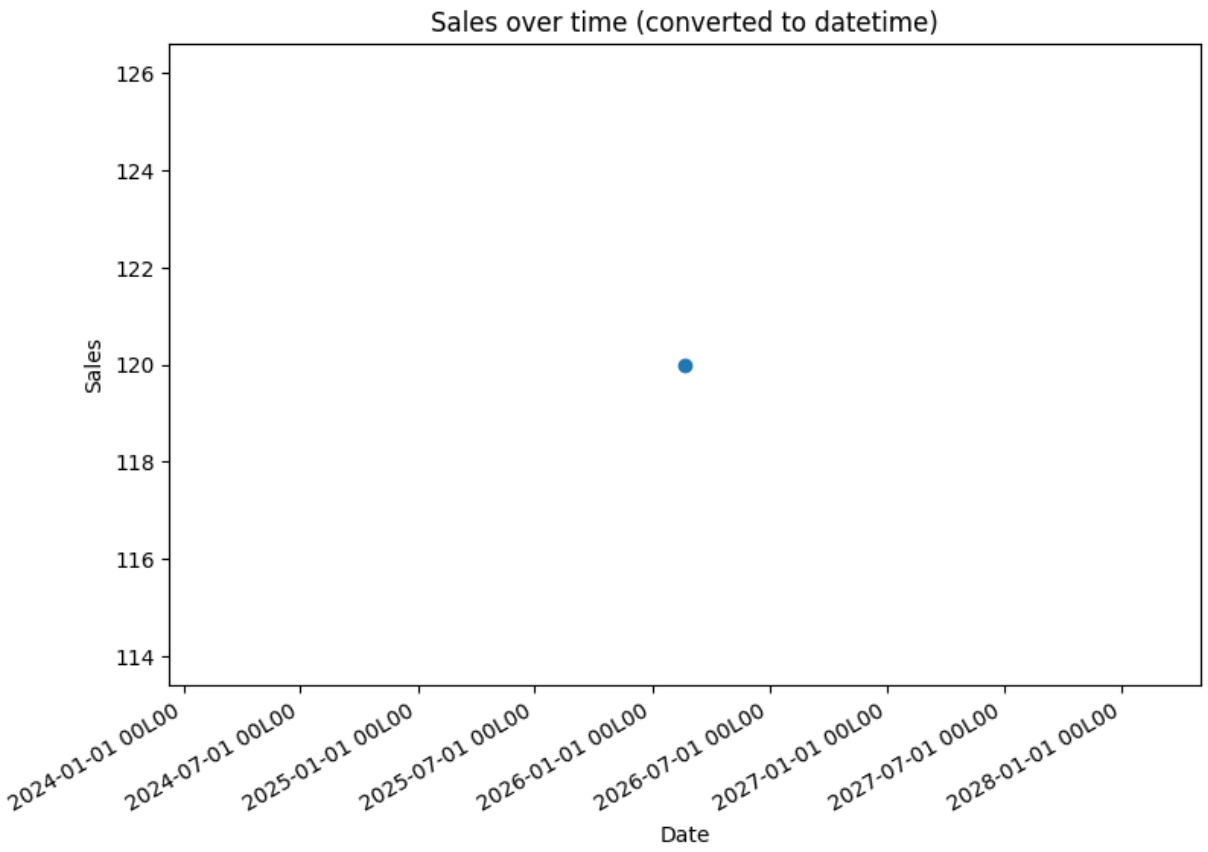
Why Pandas Datetime Is Essential

Dates stored as strings are just text.
Matplotlib cannot *reason* about text.

Pandas datetime:

- understands time order
- supports resampling
- enables rolling windows

Convert to datetime




```
import matplotlib.dates as mdates
import pandas as pd
import matplotlib.pyplot as plt

df = pd.DataFrame({
"date":[
"2026-02-20",
"2026-02-21 14:30",
"22/02/2026",
"2026-02-24T09:15:00",
"not a date"
],
"sales":[120, 150, 90, 200, 175]
})

df["date"] = pd.to_datetime(df["date"], errors="coerce", dayfirst=True)

df = df.dropna(subset=["date"]).sort_values("date")

```

```
fig, ax = plt.subplots()
ax.plot(df["date"], df["sales"], marker="o")

ax.xaxis.set_major_formatter(mdates.DateFormatter("%Y-%m-%d %HL%M"))
fig.autofmt_xdate()

ax.set_title("Sales over time (converted to datetime)")
ax.set_xlabel("Date")
ax.set_ylabel("Sales")
plt.show()

print(df)
```

Converting to datetime is like switching from labels to a real clock.

Now Pandas and Matplotlib can:

- space dates correctly
- skip weekends if needed
- compute rolling windows

2. Formatting Dates on the X-Axis

Why Formatting Matters

Raw dates often:

- overlap
- clutter
- reduce readability

The goal is human-friendly time labels.

Automatic Formatting (Recommended)

Matplotlib is smart, if dates are datetime objects, it auto-formats.

```
ax.plot(df.index, df["sales"])
```

But sometimes you need control.

Manual Date Formatting

```
import matplotlib.dates as mdates
```

```
2.
3. ax.xaxis.set_major_formatter(mdates.DateFormatter("%b %d"))
4. ax.xaxis.set_major_locator(mdates.DayLocator(interval=7))
```

Date formatting is like **choosing how detailed a calendar should be**.

3. Rotating Ticks (Avoiding Overlap)

Why Rotate Ticks?

Time labels are long.
Horizontal text often overlaps.

Rotate X-Ticks

```
1. plt.xticks(rotation=45)
```

Or OO-style:

```
1. for label in ax.get_xticklabels():
2.     label.set_rotation(45)
```

Rotating ticks is like tilting your head to read crowded text, but done once for the viewer.

4. Plotting Rolling Averages

Why Rolling Averages Exist

Raw time series data is often:

- noisy
- volatile
- hard to interpret

Rolling averages:

- smooth fluctuations
- reveal trends
- reduce noise

How Rolling Averages Work

A **7-day rolling average** means:

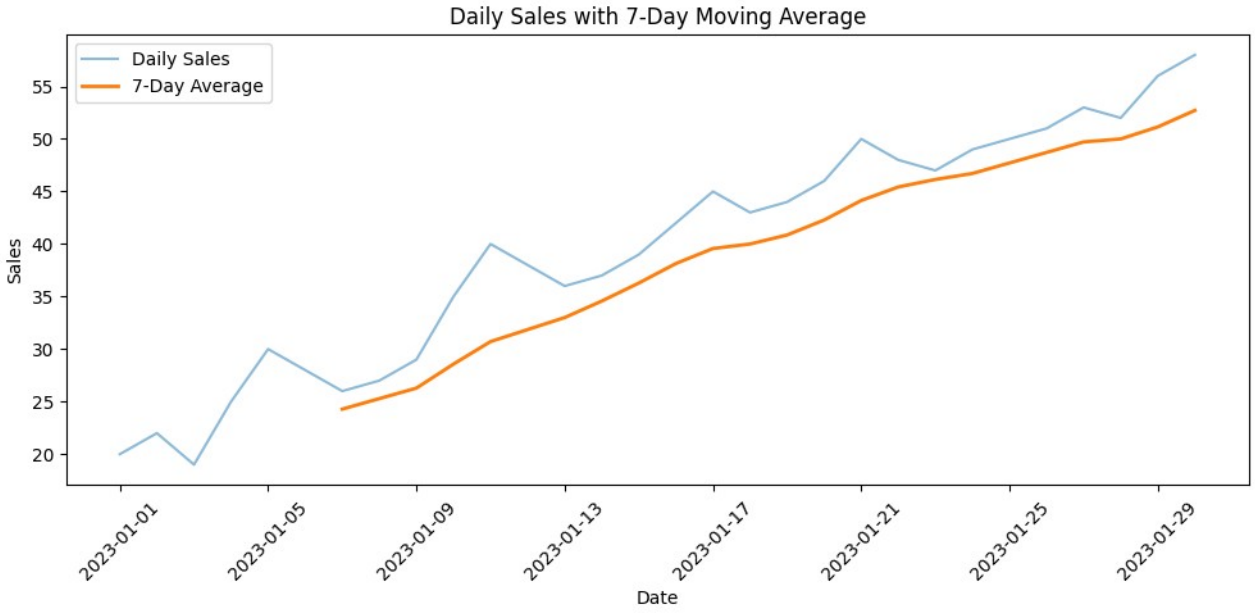
> "At each day, average the last 7 days."

Compute Rolling Average (Pandas)

```
df["rolling_7"] = df["sales"].rolling(7).mean()
```

This integrates Pandas computation with Matplotlib visualization.

Practical: Time Series with 7-Day Moving Average




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

dates = pd.date_range("2023-01-01", periods=30)
sales = [20, 22, 19, 25, 30, 28, 26, 27, 29, 35,
         40, 38, 36, 37, 39, 42, 45, 43, 44, 46,
         50, 48, 47, 49, 51, 53, 52, 54, 56, 58]

df = pd.DataFrame({"date": dates, "sales": sales})
df = df.set_index("date")

df["rolling_7"] = df["sales"].rolling(7).mean()

```

```
fig, ax = plt.subplots(figsize=(10, 5))

ax.plot(df.index, df["sales"], label="Daily Sales", alpha=0.5)
ax.plot(df.index, df["rolling_7"], label="7-Day Average", linewidth=2)

ax.set_title("Daily Sales with 7-Day Moving Average")
ax.set_xlabel("Date")
ax.set_ylabel("Sales")

ax.legend()
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

Lesson 6 Takeaways

- Always convert dates to datetime
- Use Pandas for time calculations
- Let Matplotlib format dates when possible
- Rolling averages reveal trends

Lesson 7: Plotting With Pandas

Pandas includes built-in plotting because:

Most data analysis starts with DataFrames.

Pandas plotting is a convenience layer on top of Matplotlib.

Big Idea: Pandas Is Not a Replacement for Matplotlib

Pandas plotting:

- uses Matplotlib internally
- creates Figures and Axes for you
- is quick and readable

Matplotlib:

- gives fine-grained control

The real power comes from **combining both**.

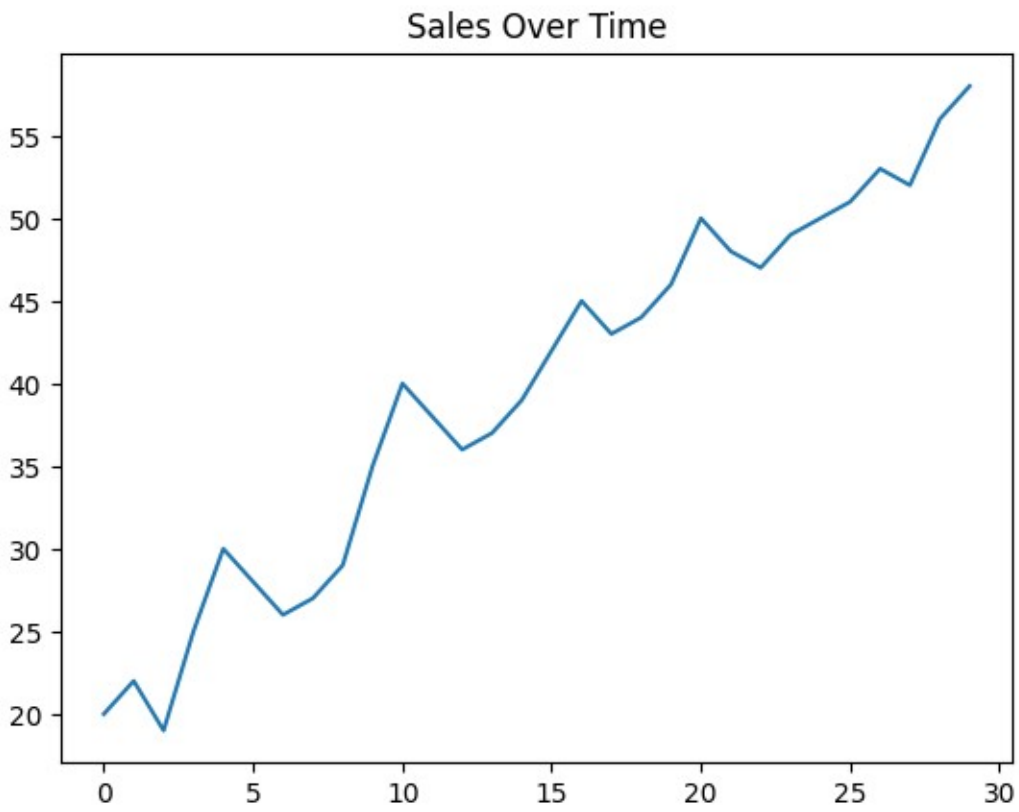
1. Using `df.plot()`

Why `df.plot()` Exists

It lets you go from:

"I have data"
to
"I have a plot"
in one line.

Line Plot from DataFrame




```
sales = [
20, 22, 19, 25, 30, 28, 26, 27, 29, 35,
40, 38, 36, 37, 39, 42, 45, 43, 44, 46,
50, 48, 47, 49, 50, 51, 53, 52, 56, 58
]
df = pd.DataFrame({"sales":sales})
df["sales"].plot(title="Sales Over Time")
```

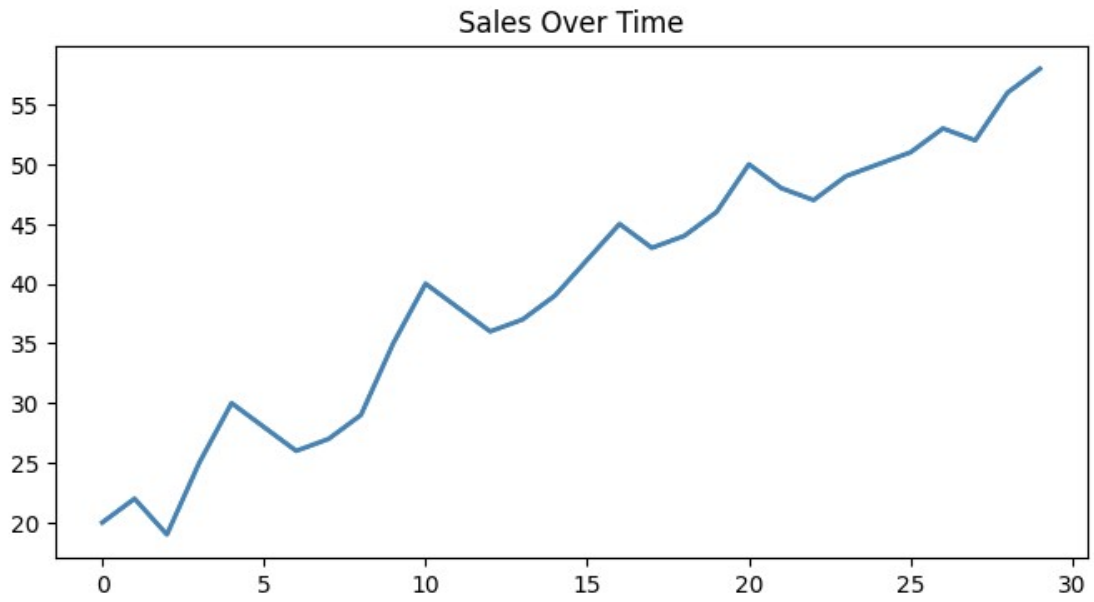
```
plt.show()

df["sales"].plot(title="Sales Over Time")
plt.show()
```

This automatically:

- creates a Figure
- uses index for x-axis
- labels axes

2. Styling Pandas Plots

Pandas accepts many Matplotlib styling arguments.




```
df["sales"].plot(
    color="steelblue",
    linewidth=2,
    figsize=(8, 4)
)
```

You can also apply global styles:

```
plt.style.use("seaborn-v0_8")
```

3. Combining Pandas + Matplotlib Customization

This is the **professional workflow**.

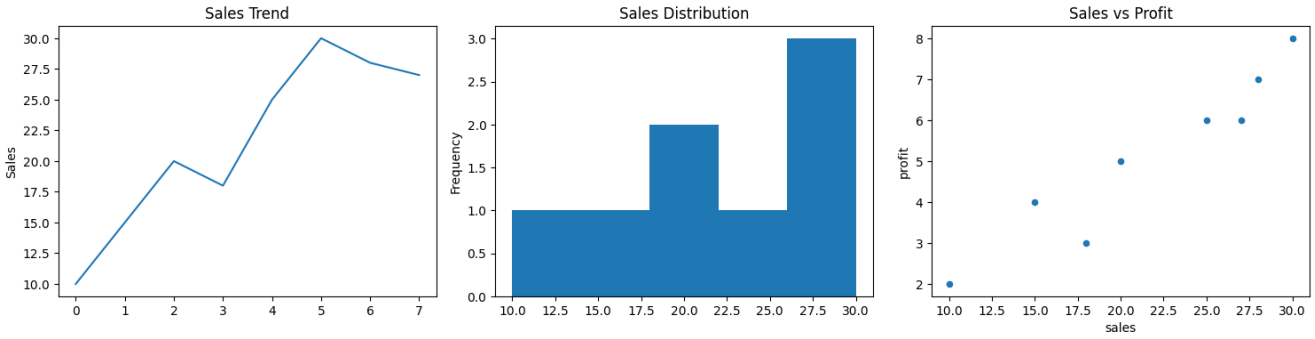
Step-by-Step Pattern

```
ax = df.plot()
ax.set_title("Custom Title")
ax.set_xlabel("Date")
ax.set_ylabel("Sales")
ax.grid(True)
```

Why this works:

- Pandas creates the Axes
- Matplotlib customizes it

Practical: Line + Histogram + Scatter from a DataFrame




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

data = {
    "sales": [10, 15, 20, 18, 25, 30, 28, 27],
    "profit": [2, 4, 5, 3, 6, 8, 7, 6]
}

df = pd.DataFrame(data)

fig, axes = plt.subplots(1, 3, figsize=(15, 4))

df["sales"].plot(ax=axes[0], title="Sales Trend")
axes[0].set_ylabel("Sales")
```

```
15.
16. df["sales"].plot(kind="hist", bins=5, ax=axes[1], title="Sales Distribution")
17.
18. df.plot(
19.     kind="scatter",
20.     x="sales",
21.     y="profit",
22.     ax=axes[2],
23.     title="Sales vs Profit"
24. )
25.
26. plt.tight_layout()
27. plt.show()
```

Lesson 7 Takeaways

- Pandas plotting is built on Matplotlib
- `df.plot()` is great for quick exploration
- Matplotlib adds polish and control
- The best charts use **both together**

At this point, learners can:

- visualize time series correctly
- smooth noisy data
- move seamlessly between Pandas and Matplotlib
- create clean, readable, professional plots

Lesson 8: Customizing Ticks & Scales

Ticks and scales are not cosmetic details.
They control how numbers are perceived, and small choices here can dramatically change interpretation.

A great chart can still fail if:

- tick labels are unclear
- scales hide patterns
- numbers are hard to read

This lesson teaches you to **control how viewers read your data**.

What Are Ticks and Scales?

- **Ticks** = reference points along the axes
- **Tick labels** = the numbers/text shown
- **Scale** = how data values map to space

Think of it this way:

> Ticks are the markings on a ruler.
> The scale is how that ruler is stretched or compressed.

1. Changing Tick Labels

Why Change Tick Labels?

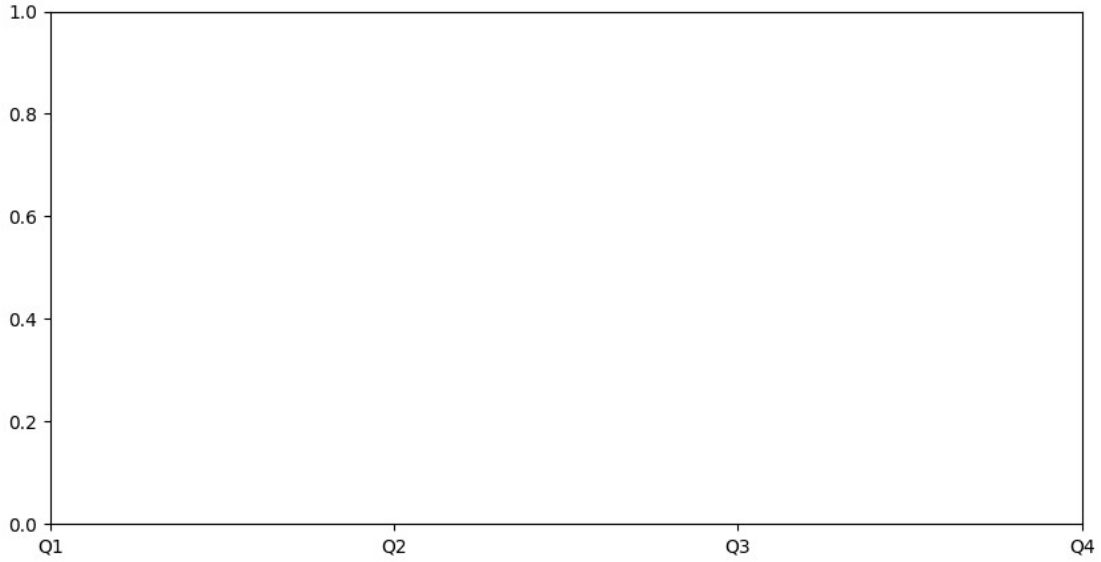
Default tick labels:

- may be too dense
- too technical

- not meaningful for humans

Your goal is human-friendly interpretation.

Basic Tick Label Control




```
1. ax.set_xticks([0, 2, 4, 6])
2. ax.set_xticklabels(["Q1", "Q2", "Q3", "Q4"])
```

Think of it this way:

This is like replacing **raw coordinates** with **street names**.

When This Is Useful

- financial quarters
- categories disguised as numbers
- stages or milestones

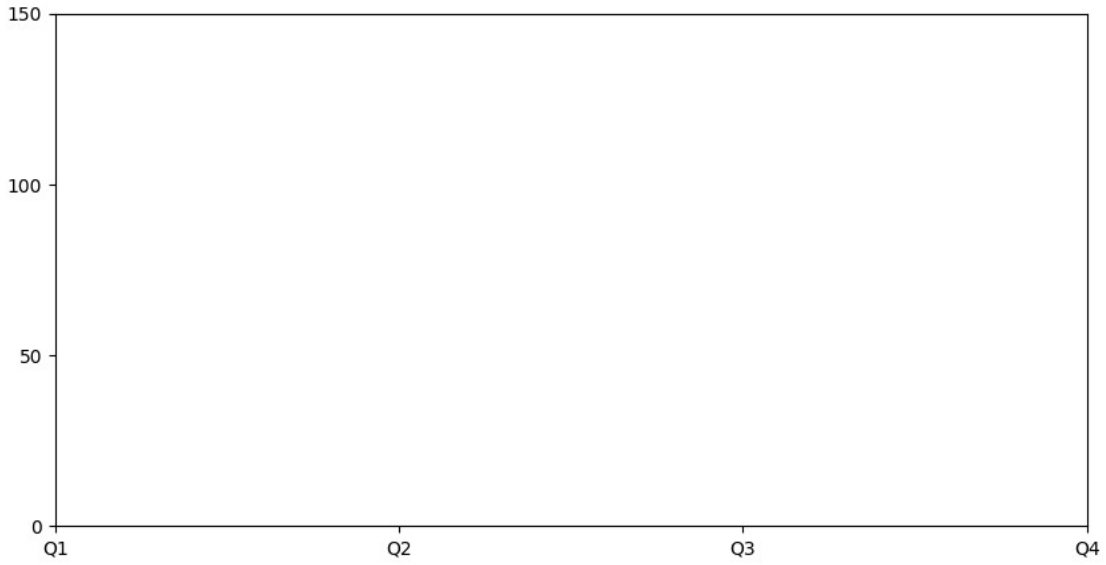
2. Using Custom Ticks

Why Custom Ticks Matter

Too many ticks = clutter
Too few ticks = loss of precision

Custom ticks strike a balance.

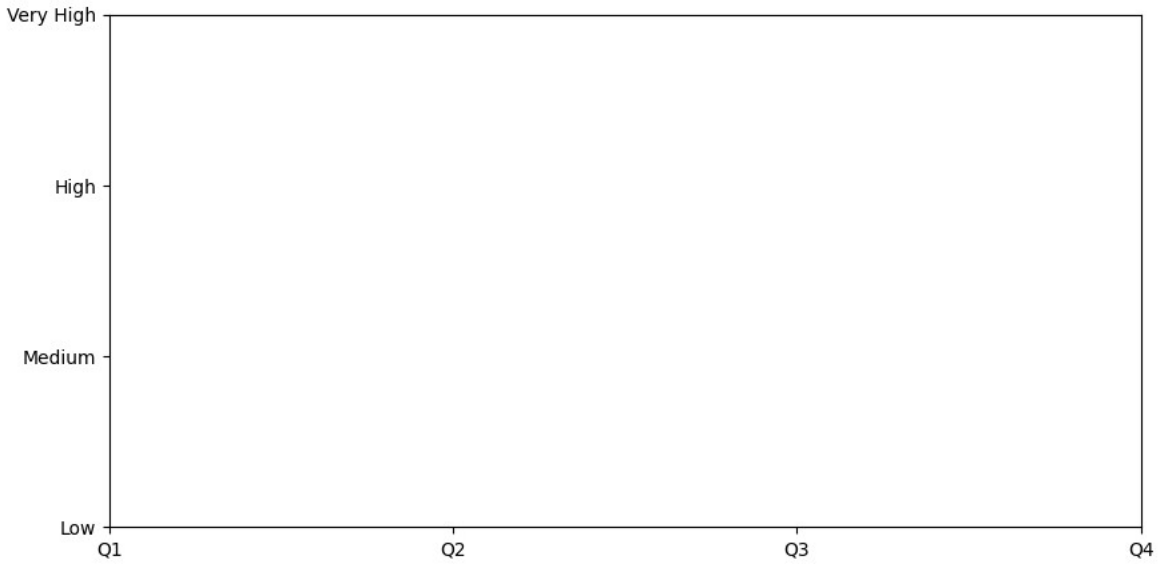
Set Custom Tick Positions




```
1. ax.set_yticks([0, 50, 100, 150])
```

This forces Matplotlib to show only those values.

Combine With Labels



```
1. ax.set_yticklabels(["Low", "Medium", "High", "Very High"])
```

This turns numeric data into **semantic meaning**.

3. Log Scale Plots

Why Log Scales Exist

Some data spans orders of magnitude:

- population sizes
- file sizes
- income distributions
- scientific measurements

A linear scale hides small values.

Linear vs Log (Analogy)

Linear scale:

> Treats 1 ⟶ 10 the same as 90 ⟶ 100

Log scale:

> Treats multiplicative change equally
> (1 ⟶ 10 ⟶ 100 ⟶ 1000)

Apply Log Scale

```
1. ax.set_yscale("log")
```

Now:

- each step = ×10
- exponential growth becomes readable

⚠ Important Warning

Never use log scales without:

- clear labeling
- explanation

Log scales change perception.

4. Scientific Notation

Why Scientific Notation Is Needed

Large or tiny numbers:

- clutter tick labels
- reduce readability

Scientific notation compresses values.

Enable Scientific Notation

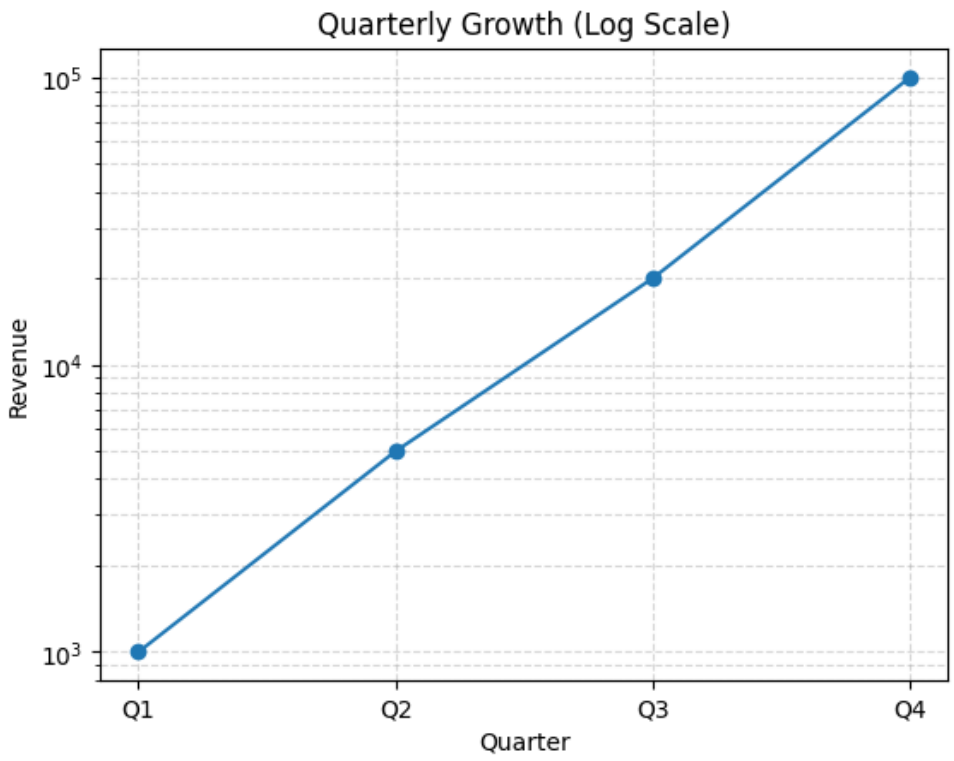
```
1. ax.ticklabel_format(style="scientific", axis="y", scilimits=(0, 0))
```

This shows values like:

```
2e6 instead of 1,200,000
```

> Scientific notation is **shorthand for big numbers**, like saying "1M" instead of "1,000,000".

Practical: Custom Tick Formatting




```python
import matplotlib.pyplot as plt

x = [1, 2, 3, 4]
y = [1000, 5000, 20 000, 100 000]

fig, ax = plt.subplots()

ax.plot(x, y, marker="o")

ax.set_yscale("log")
ax.set_xticks(x)
ax.set_xticklabels(["Q1", "Q2", "Q3", "Q4"])

ax.set_title("Quarterly Growth (Log Scale)")
ax.set_xlabel("Quarter")
ax.set_ylabel("Revenue")

ax.grid(True, which="both", linestyle="--", alpha=0.5)

plt.show()
```

Lesson 8 Takeaways

- Ticks guide interpretation
- Custom labels improve clarity
- Log scales reveal hidden patterns
- Scientific notation improves readability
- Axis design is data communication, not decoration

Lesson 9: Saving & Exporting Figures

A visualization is not finished until it is:

- shared
- embedded
- published
- printed

Saving figures correctly ensures they look **professional everywhere**.

Screens vs Print vs Web

Different outputs require different formats.

Use Case	Best Format
Web	PNG, SVG
Reports	PDF
Presentations	PNG
Editing later	SVG

1. Saving Figures

Basic Save

```
1. plt.savefig("figure.png")
```

This saves exactly what you see.

File Formats Explained

- **PNG** ⟶ raster, great for web
- **JPG** ⟶ compressed, smaller size
- **SVG** ⟶ vector, infinitely scalable
- **PDF** ⟶ print-ready, professional

2. DPI Settings

What Is DPI?

DPI = dots per inch
Controls image **sharpness**.

High-Resolution Save

```
1. plt.savefig("figure.png", dpi=300)
```

- 72 DPI ⟶ screens
- 300 DPI ⟶ print

> DPI is like camera resolution: higher = sharper.

3. Transparent Backgrounds

Useful when:

- placing figures on slides
- overlaying on colored backgrounds

```
1. plt.savefig("figure.png", transparent=True)
```

4. Publication-Quality Figures

Professional figures require:

- correct size
- consistent fonts
- high DPI
- clean layout

Control Figure Size

```
fig, ax = plt.subplots(figsize=(8, 5))
```

This ensures consistency across documents.

Always Use Tight Layout

```
plt.tight_layout()
```

Prevents clipping in saved files.

Practical: Save a Figure in Multiple Formats

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]

fig, ax = plt.subplots(figsize=(6, 4))

ax.plot(x, y, marker="o")
ax.set_title("Export Example")
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.grid(True)

plt.tight_layout()

plt.savefig("plot.png", dpi=300)
plt.savefig("plot.pdf")
plt.savefig("plot.svg", transparent=True)

plt.show()
```

Lesson 9 Takeaways

- Choose the right format for the right medium
- DPI controls clarity
- SVG & PDF are ideal for publication
- Transparent backgrounds increase flexibility
- Saving is part of visualization design

At this point, you can:

- control perception through scales and ticks
- create readable, honest axes
- export charts for any platform
- produce publication-quality figures

Optional Advanced Topics

For learners who want to go beyond the basics and explore modern, advanced visualization techniques.

These topics are not required to be effective with Matplotlib but they open doors to:

- cleaner aesthetics
- higher-level abstractions
- interactivity
- advanced storytelling

Think of this section as the "next level" after mastering the fundamentals.

1. Seaborn Introduction (Statistical Visualization Made Easy)

What Is Seaborn?

Seaborn is a high-level visualization library built on top of Matplotlib.

Key idea:

> Seaborn does not replace Matplotlib, it uses it.

Matplotlib is the engine.
Seaborn is the polished interior.

Why Seaborn Exists

Matplotlib gives you control, but requires effort.
Seaborn gives you:

- beautiful default styles
- built-in statistical plots
- automatic grouping and aggregation

Think of it this way:

Matplotlib is raw paint and brushes.
Seaborn is pre-mixed colors and design templates.

What Seaborn Is Best At

- distribution plots
- category comparisons
- regression plots
- pairwise relationships

Example

```
import seaborn as sns
import pandas as pd

sns.set_theme(style="darkgrid")
data = {
"date": pd.date_range(start="2024-01-01", periods=10),
"sales": [120, 150, 170, 160,180, 200, 220, 210,230,250]
}

df=pd.DataFrame(data)
sns.lineplot(data=df, x="date", y="sales")

plt.title("Daily Sales Trend")
plt.xlabel("Date")
plt.ylabel("Sales")
plt.xticks(rotation=45)

plt.tight_layout()
```

```
plt.show()
```

Seaborn:

- formats the chart
- applies consistent style
- handles legends automatically

You can still customize with Matplotlib afterward.

2. Heatmaps (Visualizing Patterns in Grids)

What a Heatmap Really Shows

A heatmap visualizes **values as colors in a grid**.

It answers:

> "Where are the highs, lows, and patterns?"

When Heatmaps Are Useful

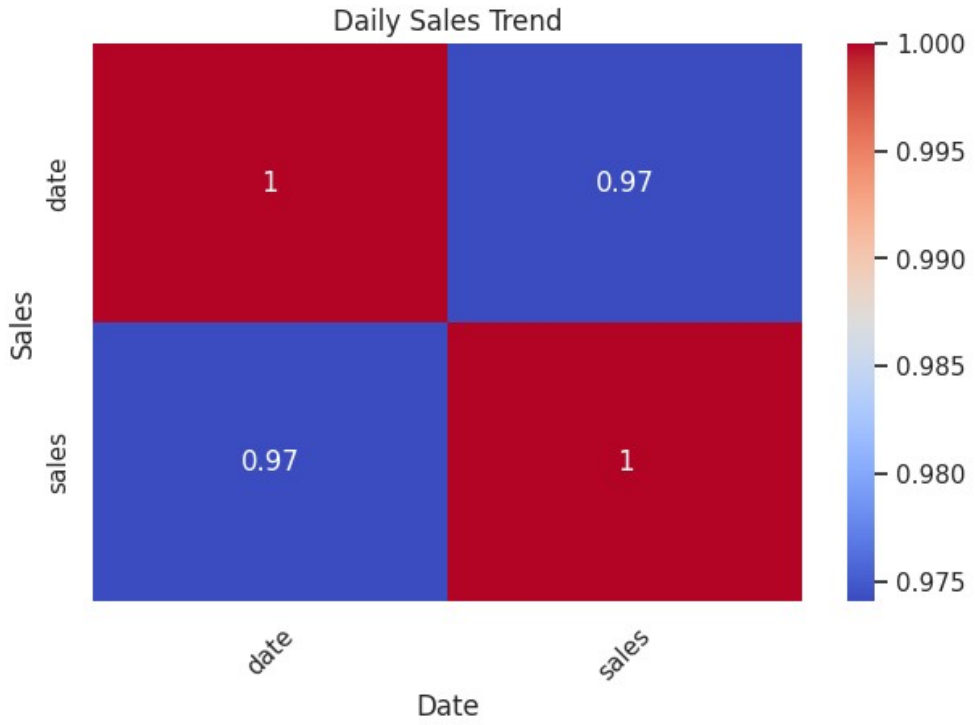
- correlation matrices
- calendar data (days × months)
- activity intensity

- confusion matrices

A heatmap is like a weather map,
colors instantly reveal hot and cold regions.

Example (Correlation Heatmap)

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



Why heatmaps matter:

- patterns are visible instantly
- numbers alone cannot do this

3. 3D Plots (Use Sparingly, Use Carefully)

What 3D Plots Are

3D plots add a third axis to visualization.

They can show:

- surfaces

- trajectories
- spatial relationships

⚠ Important Warning

3D plots:

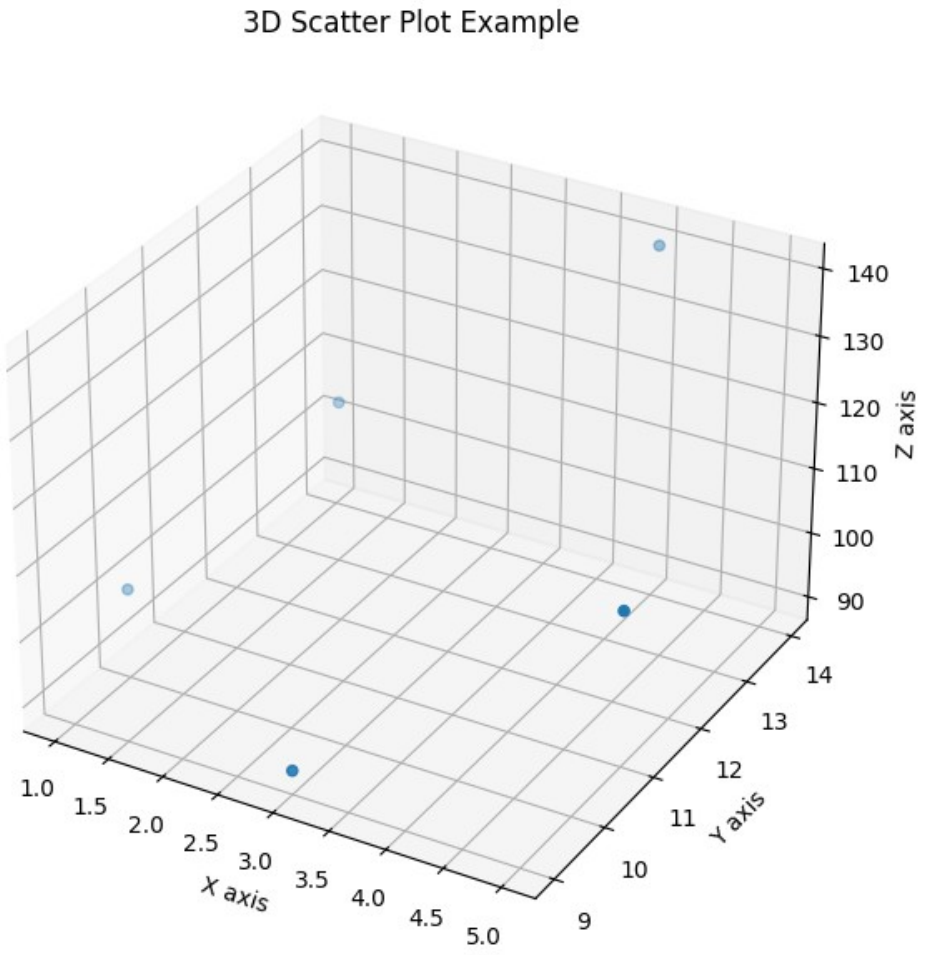
- look impressive
- are often harder to interpret

Use them only when the third dimension adds real meaning.

Think of it this way

3D plots are like aerial photos,
powerful, but easy to misread without context.

Example: 3D Scatter



```
1. import matplotlib.pyplot as plt
2. from mpl_toolkits.mplot3d import Axes3D
3.
4. x = [1,2,3,4,5]
5. y = [10, 12, 9,14,11]
6. z = [100,120,90,140,110]
7. fig = plt.figure(figsize=(8, 6))
8. ax = fig.add_subplot(111, projection="3d")
9.
10. ax.scatter(x, y, z)
11. ax.set_title("3D Scatter Plot Example")
12. ax.set_xlabel("X axis")
13. ax.set_ylabel("Y axis")
14. ax.set_zlabel("Z axis")
15.
16. plt.tight_layout()
17. plt.show()
```

Good for:

- scientific data
- simulations
- spatial modeling

4. Animation with Matplotlib (Data Over Time)

Why Animation Exists

Some insights appear only when data moves.

Animations show:

- change over time
- progression
- evolution

Use Cases

- stock prices evolving
- simulations
- training models

- sensor data

Think of it this way:

Static plots are photographs.
Animations are videos.

Example Concept (Simplified)

```
1. from matplotlib.animation import FuncAnimation
```

Animations require:

- frame updates
- redraw logic
- performance awareness

They are powerful, but advanced.

5. Interactive Plots: Plotly vs Matplotlib

Static vs Interactive Plots

Matplotlib:

- static images
- ideal for reports, PDFs, publications

Plotly:

- interactive
- hover tooltips
- zoom & pan
- web-friendly

Think of it this way:

Matplotlib charts are printed posters.
Plotly charts are interactive dashboards.

Comparison

Feature	Matplotlib	Plotly
Interactivity	✗	✓
Static export	✓	⚠
Web dashboards	✗	✓
Publication-ready	✓	⚠
Learning curve	Low	Medium

When to Use Each

Use **Matplotlib** when:

- writing reports
- publishing research
- exporting figures

Use **Plotly** when:

- building dashboards
- exploring data interactively
- sharing results online

Optional Section Takeaways

- Seaborn simplifies statistical plots
- Heatmaps reveal patterns instantly
- 3D plots should be used carefully
- Animation adds time as a dimension
- Plotly complements Matplotlib, it doesn't replace it

Lesson 10: Mini Project (Capstone Experience)

This lesson is not about learning *new syntax*.
It's about using what you already know to tell a clear story with data.

Up to now, you've learned:

- how Matplotlib works
- how to choose the right plot
- how to style charts professionally
- how to work with time series
- how to annotate insights
- how to export figures

Now you'll combine all of that into one complete visualization project.

Why a Mini Project Matters

Anyone can follow a tutorial.
What matters is whether you can:

- choose the right chart
- decide what deserves emphasis

- arrange visuals logically
- communicate insight clearly

This project simulates real-world work:

- analyst reports
- dashboards
- presentations
- portfolio visuals

Think of it this way:

This is the "final presentation," not the homework.
You're no longer practicing brush strokes , you're painting a full picture.

Choose ONE Project Path

Learners choose *one* option based on interest and comfort level.
All options are equally valid.

Option 1: Dashboard-Style Figure (4 Subplots)

Goal

Create a single figure containing four related plots arranged in a grid.

Why This Matters

Dashboards are everywhere:

- business intelligence
- analytics tools
- monitoring systems

This option teaches:

- subplot layout
- visual hierarchy
- comparison across charts

Suggested Structure (Example)

```
[ Line Plot ]   [ Bar Chart ]
[ Histogram ]  [ Scatter Plot ]
```

Example Use Case

Sales dataset:

- top-left → sales over time
- top-right → sales by category
- bottom-left → distribution of daily sales
- bottom-right → sales vs profit

Requirements

- Use `plt.subplots()`
- Add titles to each subplot
- Apply consistent styling
- Use `tight_layout()`
- At least one annotation

What This Demonstrates

- layout control
- clarity
- comparison skills

Option 2: Recreate a Real-Life Chart

Goal

Recreate a real chart you've seen before, as closely and cleanly as possible.

Examples:

- COVID-19 case curve
- stock price trend

- population growth chart
- economic indicator

Why This Matters

Recreating charts:

- builds attention to detail
- teaches visual literacy
- shows you understand *why* the chart was designed that way

Think of it this way:

> This is like learning music by playing existing songs.
> You learn structure, timing, and style.

Requirements

- Match chart type correctly
- Include proper labels and title
- Format axes clearly
- Add annotations if present
- Use appropriate scale (linear or log)

What This Demonstrates

- plot selection judgment
- design awareness
- real-world relevance

Option 3: Visualize Your Own Dataset

Goal

Create a meaningful visualization using data you care about.

Examples:

- sales data
- weather data

- fitness logs
- study hours
- expenses

Why This Matters

Personal data:

- increases engagement
- leads to better questions
- mirrors real analysis work

This option emphasizes storytelling over complexity.

Requirements

- Load or create a dataset
- Choose an appropriate chart
- Style it clearly
- Add at least one insight annotation

What This Demonstrates

- data understanding
- decision-making
- ability to explain insights

Suggested Project Workflow (All Options)

1. **Understand the question**
    - What story am I telling?
2. **Choose the right chart(s)**
3. **Create the base plot**
4. **Customize styling**
5. **Add labels, titles, legend**
6. **Annotate insights**

7. **Adjust layout**
8. **Export the figure**

Outcome (What Learners Finish With)

By the end of Lesson 10, learners will have:

✅ A complete, polished visualization
✅ A figure suitable for reports or presentations
✅ A project they can share on LinkedIn or a portfolio
✅ Confidence using Matplotlib independently

Skills Demonstrated

Skill	Description
Visualization design	Choosing the right plot
Matplotlib mastery	Using OO API effectively
Styling	Creating clean, readable charts
Storytelling	Guiding the viewer to insights
Professional output	Export-ready figures
