Day 4 - Session 3: Matplotlib for Visualization

This notebook provides a beginner-friendly deep dive into Matplotlib, the most widely used plotting library in Python. Matplotlib allows you to create static, animated, and interactive visualizations in Python. It is fundamental for generating plots for reports, analyzing sensor trends, and presenting scientific data. We will cover how to create common plot types, customize their appearance, arrange multiple plots, and save figures, all with simple and clear examples.

1. Introduction to Matplotlib: The Plotting Canvas

Imagine Matplotlib as your drawing canvas for data. It helps you turn numbers and data into pictures (plots).

Why Use Matplotlib?

- **Visualize Data**: Turn complex numbers into easy-to-understand graphs.
- **Understand Trends**: See patterns, outliers, and relationships in your data.
- **Create Reports**: Generate high-quality plots for presentations and publications.
- **Highly Customizable**: You have a lot of control over how your plots look.
- **Foundation**: Many other plotting libraries (like Seaborn) are built on Matplotlib.

Conceptual Approach:

We'll typically `import matplotlib.pyplot` (conventionally as plt) to access most of its plotting functions. We'll also use numpy to generate some sample data easily.

```python
import matplotlib.pyplot as plt

print("--- Introduction to Matplotlib (Simplified) ---")

# Step 1: Prepare some simple data points
```

```
x = [0, 1, 2, 3]  # X-axis values
y = [1, 2, 4, 3]  # Y-axis values corresponding to each x

# Step 2: Create a basic line plot
plt.plot(x, y)  # Draw a line through the points (x[i], y[i])

# Step 3: Add title and axis labels for clarity
plt.title("Simple Line Plot")      # Title at the top of the graph
plt.xlabel("X values")             # Label below the horizontal axis
plt.ylabel("Y values")             # Label beside the vertical axis

# Step 4: Show the plot window
plt.show()  # Always call this to display the plot (especially in scripts or
notebooks)
```

2. Basic Plotting Types

Matplotlib supports a wide variety of plot types. Here, we'll focus on the most common ones for scientific and engineering data.

2.1 Line Plots (plt.plot())

Line plots are used to show trends over continuous data, often time or a sequence.

`plt.plot()`: How to Draw a Line Plot

- **Purpose:** Draws lines and/or markers to represent data points.
- **Syntax:** `plt.plot(x_values, y_values, [fmt], **kwargs)`
    - **x_values:** (Optional) Data for the horizontal axis. If not given, uses `0, 1, 2…` as default.
    - **y_values:** Data for the vertical axis.
    - **fmt:** (Optional) Format string like 'ro-' (red circles, solid line).
    - ****kwargs:** Other options like color, linestyle, marker, label.

Python

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

print("--- Basic Plotting: Line Plots (Simplified) ---")

# ---------------- Example 1: Simple Temperature Line Plot ----------------

# Time in seconds
time = np.array([0, 1, 2, 3, 4, 5])

# Corresponding temperature readings
temperature = np.array([20.1, 20.5, 21.0, 20.8, 21.5, 21.2])

# Plot the temperature vs time
plt.plot(time, temperature)

# Add a title and axis labels
plt.title("Temperature Over Time")
plt.xlabel("Time (seconds)")
plt.ylabel("Temperature (°C)")

# Optional: Show a background grid for better readability
plt.grid(True)

# Display the plot
plt.show()

# ---------------- Example 2: Multiple Lines on One Plot ----------------

# Pressure readings over time
pressure = np.array([100.1, 100.3, 100.0, 99.8, 100.2, 100.5])

# Humidity readings over time
humidity = np.array([55, 56, 54, 57, 55, 56])

# Plot pressure with a dashed line and blue color
plt.plot(time, pressure, label='Pressure (kPa)', color='blue', linestyle='--')

# Plot humidity with circle markers and green color
plt.plot(time, humidity, label='Humidity (%)', color='green', marker='o')
```

```
# Add title and labels
plt.title("Pressure and Humidity Over Time")
plt.xlabel("Time (seconds)")
plt.ylabel("Sensor Readings")

# Add a legend to identify each line
plt.legend()

# Add grid
plt.grid(True)

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

2.2 Scatter Plots (plt.scatter())

Scatter plots are used to show the relationship between two numerical variables. Each point represents a single observation.

plt.scatter(): How to Draw a Scatter Plot

- Purpose: Draws individual points at specific (x, y) coordinates. Useful for showing correlations or clusters.
- Syntax: plt.scatter(x_values, y_values, s=None, c=None, marker=None, **kwargs)
    - x_values: Data for the horizontal axis.
    - y_values: Data for the vertical axis.
    - s: (Optional) Size of the markers (can be a single number or an array of sizes).
    - c: (Optional) Color of the markers (can be a single color or an array of colors).
    - marker: (Optional) Type of marker (e.g., 'o' for circle, 'x' for x).

Python

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

print("\n--- Basic Plotting: Scatter Plots (Simplified) ---")

# -------- Example 1: Simple Scatter Plot --------
# X-axis: Flow Rate (LPM), Y-axis: Motor Speed (RPM)
flow_rate = np.array([0.5, 1.2, 0.8, 2.0, 1.5])  # in Liters Per Minute
motor_speed = np.array([1000, 2500, 1500, 4000, 3000])  # in RPM

# Create a basic scatter plot
plt.scatter(flow_rate, motor_speed)

# Add labels and title
plt.title("Motor Speed vs Flow Rate")
plt.xlabel("Flow Rate (LPM)")
plt.ylabel("Motor Speed (RPM)")

# Add grid for better readability
plt.grid(True)

# Show the plot
plt.show()

# -------- Example 2: Colored & Sized Scatter Plot --------
# Efficiency values (used to control size and color of points)
efficiency = np.array([0.8, 0.95, 0.7, 0.9, 0.85])  # Scale: 0 to 1

# Create a scatter plot with:
# - size of points proportional to efficiency
# - color of points based on efficiency using a color map
plt.scatter(
    flow_rate, motor_speed,
    s=efficiency * 200,        # Size of each point
    c=efficiency,              # Color of each point
    cmap='viridis',            # Color map for color scale
    alpha=0.7                  # Make points slightly transparent
)

# Add title and axis labels
plt.title("Motor Speed vs Flow Rate (Colored by Efficiency)")
plt.xlabel("Flow Rate (LPM)")
plt.ylabel("Motor Speed (RPM)")

# Add a color bar to explain color coding
plt.colorbar(label="Efficiency")

# Add grid
plt.grid(True)

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

2.3 Bar Plots (plt.bar())

Bar plots are used to compare categorical data, showing the quantity or frequency for different categories.

plt.bar(): How to Draw a Bar Plot

- Purpose: Draws vertical bars (or horizontal plt.barh()) to show quantities for different categories.
- Syntax: plt.bar(x_categories, height_values, width=0.8, **kwargs)
    - x_categories: The labels for each bar (e.g., names of sensors, error types).
    - height_values: The height of each bar (the quantity/value).
    - width: (Optional) Width of the bars.

Python
```python
import matplotlib.pyplot as plt

print("\n--- Basic Plotting: Bar Plots (Simplified) ---")

# -------- Example 1: Vertical Bar Plot --------
# Data: Number of faults for each type of device
```

```python
device_types = ['Sensor', 'Actuator', 'Control Unit', 'Power Supply']
fault_counts = [15, 8, 5, 10]

# Create a bar chart
plt.bar(device_types, fault_counts)

# Add title and axis labels
plt.title("Faults per Device Type")
plt.xlabel("Device Type")
plt.ylabel("Number of Faults")

# Show grid lines only on Y-axis (optional for readability)
plt.grid(axis='y')

# Show the plot
plt.show()

# -------- Example 2: Horizontal Bar Plot --------
# Data: How often each error code occurred
error_codes = ['E101', 'E205', 'E310', 'E404']
error_frequencies = [25, 18, 12, 30]

# Create a horizontal bar chart
plt.barh(error_codes, error_frequencies)

# Add title and axis labels
plt.title("Error Code Frequency")
plt.xlabel("Frequency")
plt.ylabel("Error Code")

# Show grid lines only on X-axis
plt.grid(axis='x')

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

2.4 Histograms (plt.hist())

Histograms are used to show the distribution of a single numerical variable. They divide the data into "bins" and count how many data points fall into each bin.

plt.hist(): How to Draw a Histogram

- Purpose: Shows the frequency distribution of continuous numerical data.
- Syntax: plt.hist(data, bins=10, range=None, density=False, **kwargs)
  - data: The numerical data you want to plot.
  - bins: (Optional) Number of bins or a sequence defining bin edges.
  - range: (Optional) The lower and upper range of the bins.
  - density: (Optional) If True, the plot shows probability density.

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

print("\n--- Basic Plotting: Histograms (Simplified) ---")

# -------- Example 1: Simple Histogram --------
# Create 100 random sensor readings around 50
sensor_readings = np.random.normal(loc=50.0, scale=2.0, size=100)

# Plot histogram: shows how many readings fall into each value range
plt.hist(sensor_readings, bins=10)  # 10 bins (value ranges)
plt.title("Sensor Readings Distribution")
plt.xlabel("Reading Value")
plt.ylabel("Count")
plt.grid(axis='y')  # Only show horizontal grid lines
plt.show()

# -------- Example 2: Histogram with More Data --------
# Simulate 500 pressure values centered around 101.3
pressure_data = np.random.normal(loc=101.3, scale=0.5, size=500)

# Plot histogram with more bins and density=True to show a smooth shape
plt.hist(pressure_data, bins=25, density=True)  # 25 finer bins, shows
probability shape
plt.title("Pressure Reading Distribution")
plt.xlabel("Pressure (kPa)")
plt.ylabel("Probability Density")
plt.grid(True)
plt.show()
```

3. Customizing Plots: Making Your Plots Look Good

Matplotlib offers extensive options to customize every aspect of your plot.

3.1 Titles and Labels (plt.title(), plt.xlabel(), plt.ylabel())

These functions add descriptive text to your plot.

Conceptual Approach:

Call these functions after plt.plot() but before plt.show().

Code Implementation: (Already demonstrated in basic examples, but here's a recap)

Python
```python
import matplotlib.pyplot as plt

print("\n--- Customizing Plots: Titles and Labels (Simplified) ---")

# Sample data for plotting
x = [1, 2, 3]
y = [4, 5, 6]

# Create a basic line plot
plt.plot(x, y)

# Add a title and axis labels
plt.title("Simple Line Plot")        # Title shown on top of the plot
plt.xlabel("X Axis")                 # Label for the X-axis
plt.ylabel("Y Axis")                 # Label for the Y-axis
```

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

3.2 Legends (plt.legend())

If you plot multiple lines or elements, a legend helps identify what each one represents. You need to provide a label argument in your plt.plot() or plt.scatter() calls.

Conceptual Approach:

- Add label='Your Label' to each plotting function.
- Call plt.legend() without arguments to display all labels.

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

print("\n--- Customizing Plots: Legends (Simplified) ---")

# Generate time points: 0 to 9
time = np.arange(0, 10)

# Two simple signals
signal_a = np.sin(time)           # Sine wave
signal_b = np.cos(time) * 0.5     # Scaled cosine wave

# Plot both signals
plt.plot(time, signal_a, label="Signal A (Sine)")
plt.plot(time, signal_b, label="Signal B (Cosine)")

# Add title and axis labels
plt.title("Signal Comparison Over Time")
plt.xlabel("Time")
plt.ylabel("Amplitude")
```

```
# Add a legend to explain which line is which
plt.legend(loc='upper right')  # Try 'best', 'upper left', etc.

# Add a grid for better readability
plt.grid(True)

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

3.3 Grids (plt.grid())

Grids help in reading values from the axes more precisely.

plt.grid(): How to Add a Grid

- Purpose: Adds a grid to the plot background.
- Syntax: plt.grid(b=None, which='major', axis='both', **kwargs)
    - b: (Optional, deprecated) Use True or False.
    - axis: Which axis to apply grid to ('x', 'y', 'both').
    - linestyle: (e.g., '--', ':').
    - alpha: Transparency (0 to 1).

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

print("\n--- Customizing Plots: Grids (Simplified) ---")

# Create X values from 0 to 10 (50 points)
x = np.linspace(0, 10, 50)

# Y values follow a parabolic trend: y = x²
y = x ** 2

# --- Example 1: Grid on Both Axes with Custom Style ---
```

```
plt.plot(x, y)
plt.title("Parabolic Curve with Grid on Both Axes")
plt.xlabel("X Value")
plt.ylabel("Y Value")

# Enable grid with dotted lines, light gray color, and 60% transparency
plt.grid(True, linestyle=':', color='gray', alpha=0.6)
plt.show()

# --- Example 2: Grid Only on the Y-Axis ---
plt.plot(x, y)
plt.title("Grid Only on Y-Axis")
plt.xlabel("X Value")
plt.ylabel("Y Value")

# Show grid only on Y-axis using dashed blue lines
plt.grid(axis='y', linestyle='--', color='blue', alpha=0.4)
plt.show()
```

3.4 Colors, Linestyles, Markers (in plt.plot())

You can control the appearance of lines and points directly in plt.plot() or plt.scatter().

Common Formatting Arguments

Argument	Description	Example (plt.plot(x, y, ...))
color	Line/marker color (e.g., 'red', '#FF5733').	color='green'
linestyle	Style of the line (e.g., '-' solid, '--' dashed, ':' dotted, '-.' dash-dot).	linestyle='--'

marker	Style of data points (e.g., 'o' circle, 'x' x-mark, '^' triangle, 's' square).	marker='o'
linewidth	Width of the line.	linewidth=2
markersize	Size of the markers.	markersize=8
alpha	Transparency (0.0 to 1.0).	alpha=0.7

Python

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

print("\n--- Customizing Plots: Colors, Linestyles, Markers ---")

# Generate 15 equally spaced values from 0 to 10
data_points = np.linspace(0, 10, 15)

# Create two signals: sine and cosine
signal1 = np.sin(data_points)
signal2 = np.cos(data_points)

# Plot Signal 1 with a purple solid line and square markers
plt.plot(
    data_points,
    signal1,
    color='purple',         # Line color
    linestyle='-',          # Solid line
    marker='s',             # Square marker
    linewidth=2,            # Line thickness
    markersize=8,           # Marker size
    label='Signal 1 (Solid + Squares)'
)

# Plot Signal 2 with a dark orange dashed line and circular markers
plt.plot(
    data_points,
    signal2,
    color='darkorange',     # Line color
    linestyle='--',         # Dashed line
    marker='o',             # Circle marker
    alpha=0.6,             # Slightly transparent
    label='Signal 2 (Dashed + Circles)'
)

# Add title and axis labels
plt.title("Signal Analysis with Custom Styles")
plt.xlabel("Time")
plt.ylabel("Amplitude")

# Add legend and grid
plt.legend()
plt.grid(True)

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

3.5 Axis Limits (plt.xlim(), plt.ylim())

You can manually set the minimum and maximum values displayed on the x and y axes.

Conceptual Approach:

Use plt.xlim(min_val, max_val) and plt.ylim(min_val, max_val) to zoom in or out on specific parts of your plot.

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

print("\n--- Customizing Plots: Axis Limits ---")

# Create time points from 0 to 95 (steps of 5)
sensor_over_time = np.arange(0, 100, 5)

# Generate random pressure values between 90 and 110
```

```
pressure_values = np.random.randint(90, 110, size=len(sensor_over_time))

# --- Full Range Plot ---
plt.plot(sensor_over_time, pressure_values, marker='o')
plt.title("Pressure Readings (Full Range)")  # Title for the plot
plt.xlabel("Time Point")                    # X-axis label
plt.ylabel("Pressure (kPa)")                # Y-axis label
plt.grid(True)
plt.show()

# --- Zoomed-In Plot ---
plt.plot(sensor_over_time, pressure_values, marker='o', color='red')
plt.title("Pressure Readings (Zoomed In)")  # More focused view
plt.xlabel("Time Point")
plt.ylabel("Pressure (kPa)")
plt.xlim(20, 60)   # Limit x-axis from 20 to 60
plt.ylim(95, 105)  # Limit y-axis from 95 to 105
plt.grid(True)     # Show grid for better readability
plt.show()
```

3.6 Styles (plt.style.use())

Matplotlib comes with several pre-defined plot styles that can quickly change the overall aesthetic of your plots.

Conceptual Approach:

Call plt.style.use('style_name') at the beginning of your script or before plotting. This applies the style globally.

Some Available Styles: 'default', 'ggplot', 'fivethirtyeight', 'seaborn-v0_8', 'dark_background', 'bmh', etc.

Python

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

print("\n--- Customizing Plots: Styles ---")

# Generate X values from 0 to 2π (100 points)
x_data = np.linspace(0, 2 * np.pi, 100)

# Y values for sine and cosine
y_sin = np.sin(x_data)
y_cos = np.cos(x_data)

# --- Try different built-in styles for better aesthetics ---
# Uncomment one at a time to apply a style
# plt.style.use('ggplot')            # Stylish grid-based plot
# plt.style.use('seaborn-v0_8')      # Smooth, polished design (Seaborn theme)
# plt.style.use('dark_background')   # Dark theme for presentations

plt.style.use('default')  # This resets to matplotlib’s default style

# --- Plot the sine and cosine curves ---
plt.plot(x_data, y_sin, label='Sine')
plt.plot(x_data, y_cos, label='Cosine')
plt.title("Trigonometric Functions (Default Style)")
plt.xlabel("Angle (radians)")
plt.ylabel("Value")
plt.legend()       # Show labels for both curves
plt.grid(True)     # Add grid lines for readability
plt.show()

# Always reset style if you're switching styles mid-notebook
plt.style.use('default')
```

4. Subplots: Multiple Plots in One Figure

Sometimes you need to display multiple related plots side-by-side or in a grid within a single figure. Matplotlib offers subplots for this.

4.1 plt.subplot(rows, cols, index) (Older, simpler for basic grids)

Conceptual Approach:

plt.subplot(rows, columns, plot_number). This function divides the figure into a grid and activates a specific plot within that grid. plot_number starts from 1, counts across rows, then down columns.

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

print("\n--- Subplots: Using plt.subplot() ---")

# Create a time axis from 0 to 10, with step size 0.1
time_steps = np.arange(0, 10, 0.1)

# Define 4 different signals
signal_a = np.sin(time_steps)                               # Pure sine wave
signal_b = np.cos(time_steps)                                    # Pure cosine
wave
signal_c  =  np.sin(time_steps)  *  np.exp(-time_steps  /  10)        #  Damped  sine
wave
signal_d = np.cos(time_steps) * np.exp(-time_steps / 10)       # Damped cosine
wave

# Set the total plot size (width, height in inches)
plt.figure(figsize=(10, 8))

# --- Subplot 1: Top-left ---
plt.subplot(2, 2, 1)  # 2 rows, 2 columns, plot 1
plt.plot(time_steps, signal_a, color='blue')
plt.title("Signal A: Sine Wave")
plt.ylabel("Amplitude")

# --- Subplot 2: Top-right ---
```

```python
plt.subplot(2, 2, 2)  # 2 rows, 2 columns, plot 2
plt.plot(time_steps, signal_b, color='red')
plt.title("Signal B: Cosine Wave")
plt.ylabel("Amplitude")

# --- Subplot 3: Bottom-left ---
plt.subplot(2, 2, 3)  # 2 rows, 2 columns, plot 3
plt.plot(time_steps, signal_c, color='green')
plt.title("Signal C: Damped Sine")
plt.xlabel("Time")
plt.ylabel("Amplitude")

# --- Subplot 4: Bottom-right ---
plt.subplot(2, 2, 4)  # 2 rows, 2 columns, plot 4
plt.plot(time_steps, signal_d, color='orange')
plt.title("Signal D: Damped Cosine")
plt.xlabel("Time")
plt.ylabel("Amplitude")

# Prevent overlap of titles and labels
plt.tight_layout()

# Show all subplots together
plt.show()
```

4.2 plt.subplots(rows, cols) (Recommended for better control)

Conceptual Approach:

This function is generally preferred because it returns both the Figure object (the overall canvas) and an Axes object (or an array of Axes objects, which are the actual plots). This gives you more control over each individual subplot.

Syntax: fig, ax = plt.subplots(nrows=1, ncols=1, figsize=None, ...)

- fig: The entire figure (canvas).
- ax: The Axes object(s) representing the individual plot(s). If nrows or ncols > 1, ax will be a NumPy array of Axes objects.

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

print("\n--- Subplots: Using plt.subplots() (Recommended) ---")

# --- Example 1: Two sensors in side-by-side plots ---

# Simulated temperature data from two sensors
temperatures_sensor1 = np.array([20, 21, 20, 22, 21])
temperatures_sensor2 = np.array([25, 24, 25, 23, 24])
time_points = np.arange(len(temperatures_sensor1))  # Time steps: 0, 1, 2, 3, 4

# Create 1 row and 2 columns of subplots
fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=(12, 5))

# --- First plot: Sensor 1 ---
ax1.plot(time_points, temperatures_sensor1, marker='o', color='blue',
label='Sensor 1')
ax1.set_title("Sensor 1 Temperature")
ax1.set_xlabel("Time")
ax1.set_ylabel("Temp (°C)")
ax1.grid(True)
ax1.legend()

# --- Second plot: Sensor 2 ---
ax2.plot(time_points, temperatures_sensor2, marker='x', color='red',
label='Sensor 2')
ax2.set_title("Sensor 2 Temperature")
ax2.set_xlabel("Time")
ax2.set_ylabel("Temp (°C)")
ax2.grid(True)
ax2.legend()

# Add a common title for the entire figure
fig.suptitle("Temperature Comparison from Two Sensors", fontsize=16)
```

```
# Adjust layout so the main title doesn’t overlap with plots
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()


# --- Example 2: Signal and its Derivative stacked vertically ---

# Create a smooth time range
time_fine = np.linspace(0, 2 * np.pi, 100)
signal = np.sin(time_fine)              # Original signal (sine)
derivative_signal = np.cos(time_fine)   # Derivative of sine is cosine

# Create 2 rows and 1 column of subplots (stacked)
fig, (ax_signal, ax_deriv) = plt.subplots(nrows=2, ncols=1, sharex=True,
figsize=(8, 7))

# Plot the original sine signal
ax_signal.plot(time_fine, signal, color='purple')
ax_signal.set_title("Original Signal: Sine Wave")
ax_signal.set_ylabel("Amplitude")
ax_signal.grid(True)

# Plot the cosine derivative signal
ax_deriv.plot(time_fine, derivative_signal, color='green')
ax_deriv.set_title("Derivative Signal: Cosine Wave")
ax_deriv.set_xlabel("Time (radians)")
ax_deriv.set_ylabel("Amplitude")
ax_deriv.grid(True)

# Adjust spacing between plots
plt.tight_layout()
plt.show()
```

5. Exporting Plots: Saving Your Figures

Once you've created a beautiful plot, you'll want to save it to a file for reports or presentations.

plt.savefig(): How to Save a Plot

- Purpose: Saves the current figure to a file.
- Syntax: plt.savefig(filename, dpi=None, format=None, ...)
    - filename: The name of the file (e.g., 'my_plot.png', 'report_figure.pdf'). Matplotlib infers format from extension.
    - dpi: (Optional) Dots per inch (resolution). Higher DPI means higher quality (e.g., 300).
    - format: (Optional) File format (e.g., 'png', 'pdf', 'svg', 'jpg').

Python
```
import matplotlib.pyplot as plt
import numpy as np
import os  # To check if the file was saved

print("\n--- Exporting Plots: Saving Figures ---")

# --- Step 1: Create some data to plot ---
x_vals = np.array([1, 2, 3, 4])
y_vals = np.array([5, 7, 6, 8])

# --- Step 2: Create a simple line plot ---
plt.plot(x_vals, y_vals, marker='o')          # Plot points with circular
markers
plt.title("Data to be Saved")                 # Add a title
plt.xlabel("X")                               # Label for X-axis
plt.ylabel("Y")                               # Label for Y-axis
plt.grid(True)                                # Add a grid to the plot

# --- Step 3: Save the plot as an image file (PNG) ---
png_filename = "data_plot.png"
plt.savefig(png_filename, dpi=300)            # Save plot with high resolution
print(f"Plot saved as PNG: '{png_filename}' -> Exists:
{os.path.exists(png_filename)}")

# --- Step 4: Save the same plot as a PDF (vector format, best for printing)
```

```
---
pdf_filename = "data_plot.pdf"
plt.savefig(pdf_filename, format='pdf')        # Save as vector-based PDF
print(f"Plot saved as PDF: '{pdf_filename}' -> Exists:
{os.path.exists(pdf_filename)}")

# --- Step 5: Show the plot (important when working in notebooks or scripts)
---
plt.show()  # This also resets the current plot in Jupyter or scripts

# --- Optional: Clear or close the figure manually if not using plt.show() ---
# plt.clf()    # Clear the figure (if making new plots)
# plt.close()  # Close the figure window (useful in loops or large apps)
```

6. Visualization for Reports (Sensor Trends Example)

Here, we'll combine various plotting techniques to visualize a simulated sensor trend, similar to what you might include in an engineering report.

Problem: Visualize temperature and pressure trends from simulated sensor data over time, highlight critical points, and provide summary statistics.

Conceptual Approach:

1. Generate realistic-looking time-series data using NumPy and Pandas.
2. Plot temperature and pressure on separate subplots but share the time axis.
3. Add titles, labels, grids, and legends.
4. Highlight specific events or thresholds on the plot.
5. Save the figure for a report.

Code Implementation:

Python

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

# --- Step 1: Generate Simple Time-Series Data ---
time = pd.date_range(start='2025-07-01 08:00', periods=20, freq='H')   # 20 hourly points
temperature = 25 + np.random.normal(0, 0.5, size=20)                  # Around 25°C
pressure = 100 + np.random.normal(0, 0.3, size=20)                    # Around 100 kPa

# Create a DataFrame
df = pd.DataFrame({'Temperature': temperature, 'Pressure': pressure}, index=time)

# --- Step 2: Define Thresholds ---
TEMP_LIMIT = 26.0
PRESSURE_HIGH = 101.0
PRESSURE_LOW = 99.0

# --- Step 3: Plot Temperature and Pressure in Subplots ---
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 6), sharex=True)

# Temperature Plot
ax1.plot(df.index, df['Temperature'], color='orange', label='Temperature')
ax1.axhline(TEMP_LIMIT, color='red', linestyle='--', label=f'Limit {TEMP_LIMIT}°C')
ax1.set_ylabel("°C")
ax1.set_title("Temperature Trend")
ax1.legend()
ax1.grid(True)

# Pressure Plot
ax2.plot(df.index, df['Pressure'], color='blue', label='Pressure')
ax2.axhline(PRESSURE_HIGH, color='red', linestyle='--', label='High Limit')
ax2.axhline(PRESSURE_LOW, color='green', linestyle='--', label='Low Limit')
ax2.set_ylabel("kPa")
ax2.set_title("Pressure Trend")
ax2.set_xlabel("Time")
```

```
ax2.legend()
ax2.grid(True)

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

7. Prospect to Seaborn

While Matplotlib is powerful, it can sometimes be verbose for common statistical plots. Seaborn is a statistical data visualization library built on top of Matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

Conceptual Approach:

Think of Seaborn as making Matplotlib "prettier" and simpler for statistical plots. It automatically handles many customizations (like colors, legends, multiple variables) that require more manual effort in raw Matplotlib.

Why Use Seaborn (in addition to Matplotlib)?

- High-Level Interface: Simpler syntax for common statistical plots.
- Aesthetically Pleasing Defaults: Plots often look better out-of-the-box.
- Integrates with Pandas: Works very well with DataFrames.
- Complex Visualizations: Easily create heatmaps, violin plots, pair plots, etc.

Installation: pip install seaborn

Table: Matplotlib vs. Seaborn (High-Level Comparison)

Feature	Matplotlib	Seaborn
Level	Low-level, foundational, provides building blocks.	High-level, statistical, builds on Matplotlib.
Control	Maximum control over every plot element.	Less granular control (often good defaults).
Complexity	Can be verbose for complex plots.	Simpler syntax for common statistical plots.
Default Style	Basic, requires customization for aesthetics.	Aesthetically pleasing defaults.
Integration	Works with NumPy arrays.	Strongly integrated with Pandas DataFrames.
Best For	Highly custom plots, plotting functions, embedded plots.	Statistical plots, quick exploratory data analysis.

Code Example (Conceptual for Seaborn):

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

print("\n--- Intro to Seaborn: Simplified Statistical Plots ---")

# Create some simple data
df = pd.DataFrame({
    'Temperature': np.random.normal(25, 2, 50),
    'Pressure': np.random.normal(100, 5, 50),
    'Device': np.random.choice(['A', 'B'], 50)
})

# 1. Scatter Plot: Compare Temperature vs Pressure, color by Device type
sns.scatterplot(data=df, x='Temperature', y='Pressure', hue='Device')
plt.title("Temp vs Pressure by Device")
plt.grid(True)
plt.show()

# 2. Histogram with KDE: View temperature distribution
sns.histplot(data=df, x='Temperature', kde=True)
plt.title("Temperature Distribution")
plt.grid(True)
plt.show()
```

8. Key Takeaways

This session provided a comprehensive introduction to Matplotlib, empowering you to create a wide variety of visualizations for your scientific and engineering data.

- **Matplotlib Fundamentals**: Understood the core plotting concepts, including plt.plot() for lines, plt.scatter() for points, plt.bar() for categories, and plt.hist() for distributions.
- **Plot Customization**: Mastered essential techniques to enhance plot aesthetics, including adding titles, axis labels, and legends; controlling colors, linestyles, and markers; setting axis limits; and using predefined plot styles.
- **Subplots** for Multi-View Analysis: Learned to arrange multiple plots within a single figure using plt.subplot() and the more flexible plt.subplots(), crucial for comparative analysis.
- **Exporting High-Quality Figures**: Gained the ability to save plots to various file formats (e.g., PNG, PDF) with controlled resolution for reports and publications.
- **Practical Visualization Workflow**: Applied a combination of Matplotlib techniques to create a detailed visualization of simulated sensor trends, demonstrating a typical use case in engineering reports.
- **Introduction to Seaborn**: Understood Seaborn as a high-level library that simplifies

statistical plotting and offers enhanced aesthetics, building upon Matplotlib's capabilities.

By mastering these Matplotlib concepts, you are now well-equipped to effectively visualize your data, communicate insights, and generate compelling figures for reports in your engineering and scientific projects.