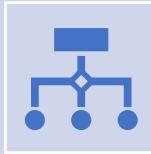


Introduction to Data Analysis

- **Foundations and Key Concepts**
- **Objectives:**
 - - Understand the fundamentals of data analysis.
 - - Differentiate between data analysis and data analytics.
 - - Learn the different types of data analysis.
 - - Understand the data analysis workflow and data types.



What is Data Analysis?



- The process of inspecting, cleaning, transforming, and modeling data.



Helps in discovering useful information, making decisions, and predicting trends.



Used in business, healthcare, and finance.

Feature	Data Analysis	Data Analytics
Focus	Understanding past data	Making future predictions
Goal	Extract insights	Improve decision-making
Example	Analyzing sales trends	Predicting future sales

Why Data Analysis?

- . **Importance in various fields:** Business, healthcare, finance, science, social sciences, etc. *Note: Applicable to almost any domain with data.*
- . **Key benefits:** Improved decision-making, identification of patterns and trends, risk reduction, increased efficiency, development of new products and services. *Note: Creates a competitive advantage.*

A blue pen with a silver clip and a blue cap lies diagonally across a white lined notebook page. A blue bar chart is overlaid on the page, featuring several vertical bars of varying heights. The text is centered over the chart.

Take 3-4 minutes
think about the
examples of direct
benefits in health
and finance?
down with your
discussion

Types of Data Analysis

- - Descriptive Analysis: What happened?
- - Diagnostic Analysis: Why did it happen?
- - Predictive Analysis: What can happen in the future?
- - Prescriptive Analysis: What actions should we take?
- Can you think of example of each of above??? Take 5 minutes and write down examples

Examples

- **Descriptive Analysis (What happened?)**
- Example: A retail store analyzes last quarter's sales data to determine that the best-selling product was a specific brand of sneakers.
- **Use Case:** Summarizing past performance, like total revenue, customer demographics, or website traffic.

Examples

- **Diagnostic Analysis (Why did it happen?)**
- Example: A company notices a sudden drop in sales and investigates factors like customer feedback, competitor activity, and economic changes to determine the cause.
- **Use Case:** Identifying reasons for business outcomes, such as a decline in customer satisfaction or increased product returns.

Examples

- **Predictive Analysis (What can happen in the future?)**
- Example: An e-commerce site uses historical purchase data to predict which products will be in high demand during the holiday season.
- **Use Case:** Forecasting future trends, such as customer churn, stock market movements, or disease outbreaks.
-

Examples

- **Prescriptive Analysis (What actions should we take?)**
- Example: A ride-sharing app uses real-time traffic and demand data to suggest optimal surge pricing and driver deployment.
- **Use Case:** Providing recommendations for action, such as optimizing marketing strategies, setting dynamic pricing, or improving customer engagement.

Process Flow of Data Analysis

- - Requirements Gathering & Planning
- - Data Collection
- - Data Cleaning
- - Data Preparation
- - Data Analysis
- - Data Interpretation & Result Summarization
- - Data Visualization

Examples of each stage

- **Requirements Gathering & Planning**
- **Example:** A retail company wants to understand customer purchasing behavior to improve its marketing strategy.
- **Process:** Define objectives, stakeholders, required data sources, and success criteria.

- **Data Collection**
- **Example:** The company gathers transaction data from its sales database, customer feedback from surveys, and website clickstream data.
- **Process:** Extract data from multiple sources like databases, APIs, web scraping, or sensors.

Data Cleaning

- Example: The collected sales data has missing customer IDs and duplicate transactions, which need to be fixed.
- Process: Handle missing values, remove duplicates, correct inconsistencies, and standardize formats.

Data Preparation

- **Example:** Convert date formats, categorize customer age groups, and merge data from different sources into a structured dataset.
- **Process:** Transform, normalize, and structure data for analysis, ensuring it's ready for processing.

Data Analysis

- **Example:** Perform exploratory data analysis (EDA) to identify purchasing patterns and segment customers based on buying behavior.
- **Process:** Apply statistical methods, machine learning models, or business intelligence tools to derive insights.

Data Interpretation & Result Summarization



Example: Analysts discover that repeat customers tend to buy more during holiday seasons,



Process: Summarize key findings, explain patterns, and relate insights to business objectives

Data Visualization

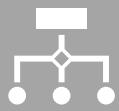
- **Example:** Create a dashboard with bar charts, heat maps, and line graphs showing sales trends and customer demographics.
 - **Process:** Use visualization tools (e.g., Tableau, Power BI, Matplotlib, Seaborn) to communicate insights effectively.



Types of Data



Structured Data Organized in tables (e.g., Excel, SQL databases).



Semi-structured Data: Partially organized (e.g., JSON, XML, CSV).



Unstructured Data: No predefined format (e.g., images, videos, social media posts).

File Formats in Data Analysis



CSV: Widely used for structured data.



JSON: Used in web applications and APIs.



XML: Common for data exchange.



Excel: Popular in business data processing.



Databases (SQL, NoSQL): Large-scale data storage.



Python Libraries for Data Analysis



NumPy : Numerical computing.

Pandas: Data manipulation and analysis.

Matplotlib: Basic visualization.

Seaborn: Advanced visualization.

Scikit-learn: Machine learning models.



We will discuss the examples of each of these libraries in the following slides

1. NumPy (Numerical Computing)

python

```
import numpy as np

# Create a NumPy array
arr = np.array([1, 2, 3, 4, 5])
print(arr * 2) # Element-wise multiplication
```

Example Output:

css

```
[ 2  4  6  8 10 ]
```

Pandas (Data Manipulation and Analysis)

python

```
import pandas as pd

# Create a DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie'], 'Age':
df = pd.DataFrame(data)
print(df)
```

	Name	Age
0	Alice	25
1	Bob	30
2	Charlie	35

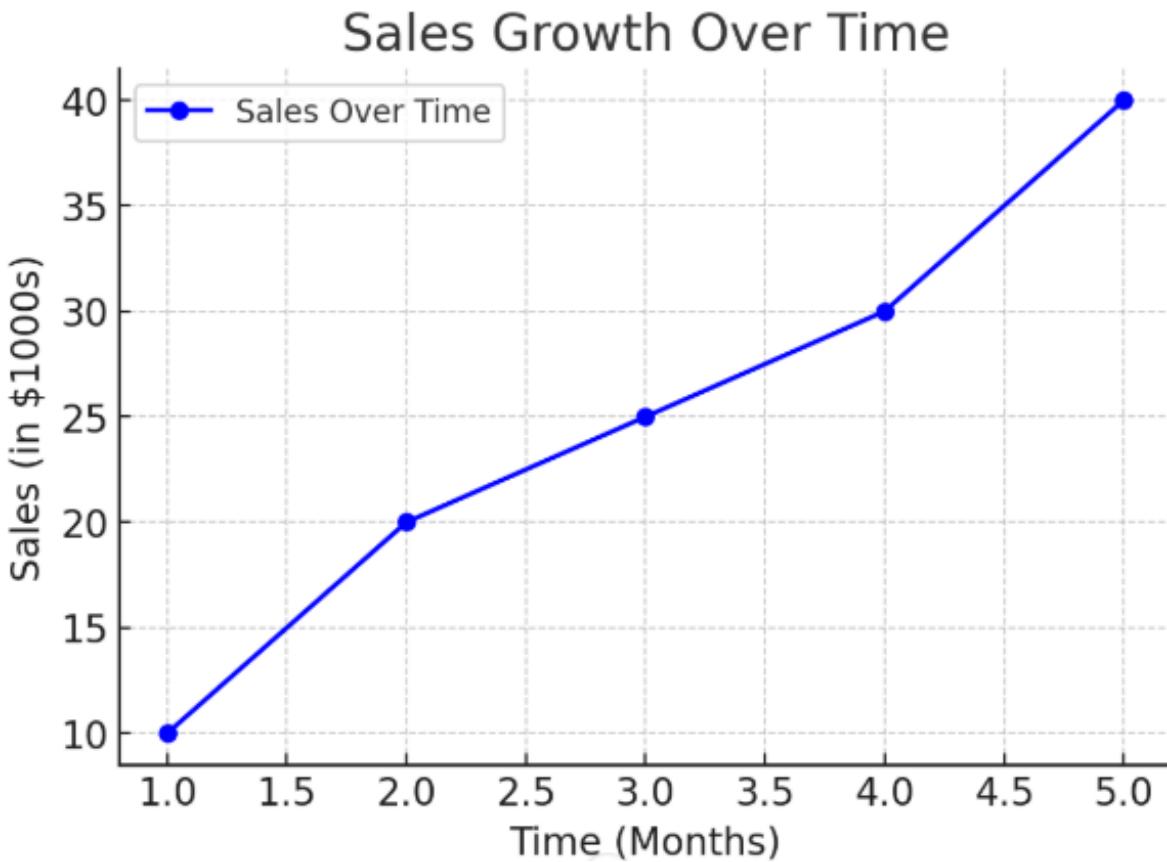
Matplotlib

```
import matplotlib.pyplot as plt

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

plt.plot(x, y, marker='o')
plt.title("Basic Line Plot")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.show()
```

Output: A simple line plot with points.



Example of line chart

5. Scikit-learn (Machine Learning Models)

python

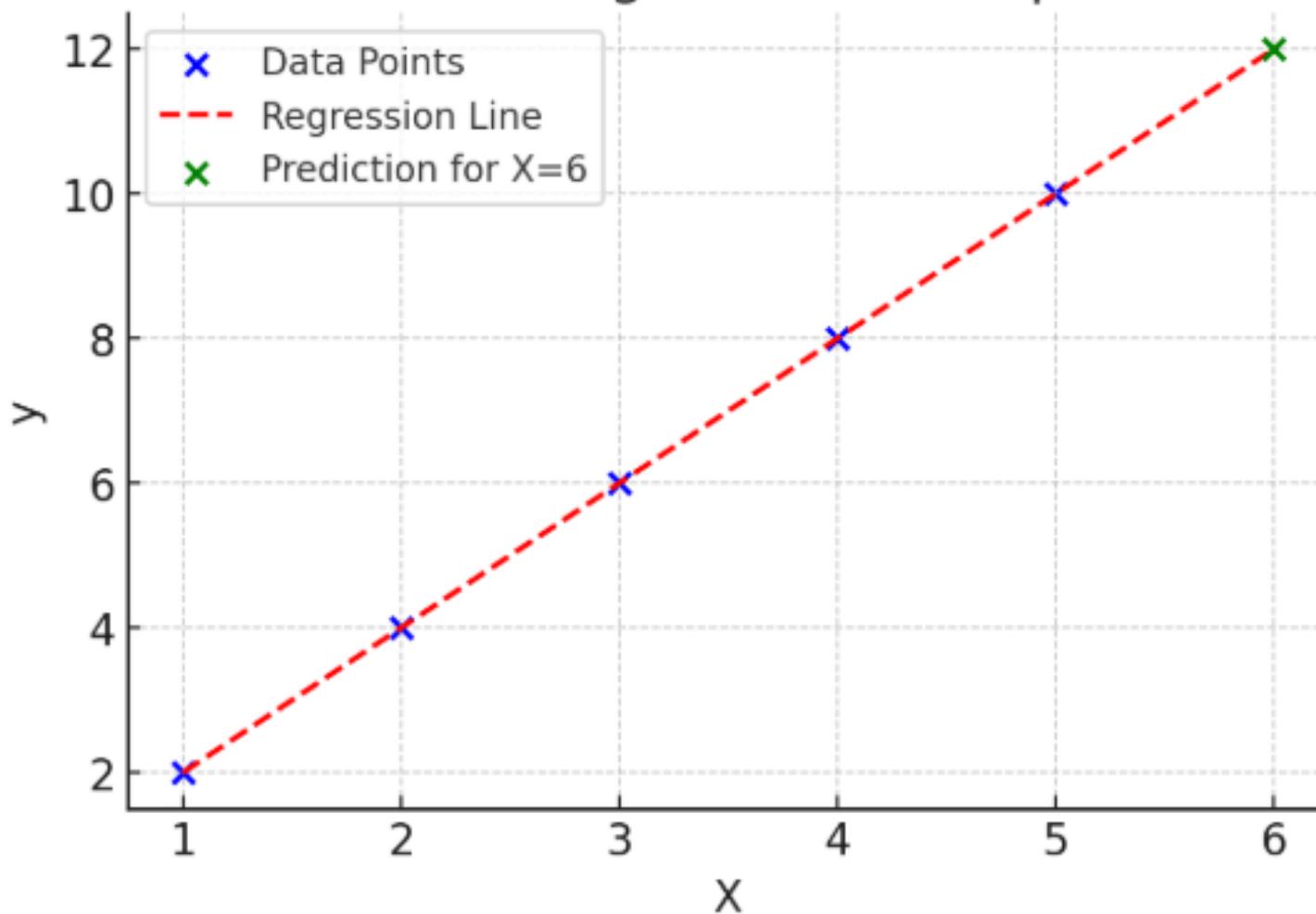
```
from sklearn.linear_model import LinearRegression
import numpy as np

# Sample data
X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)
y = np.array([2, 4, 6, 8, 10])

# Create and train model
model = LinearRegression()
model.fit(X, y)

# Make a prediction
print(model.predict([[6]])) # Predicts for X=6
```

Linear Regression Example



4. Seaborn (Advanced Visualization)

python

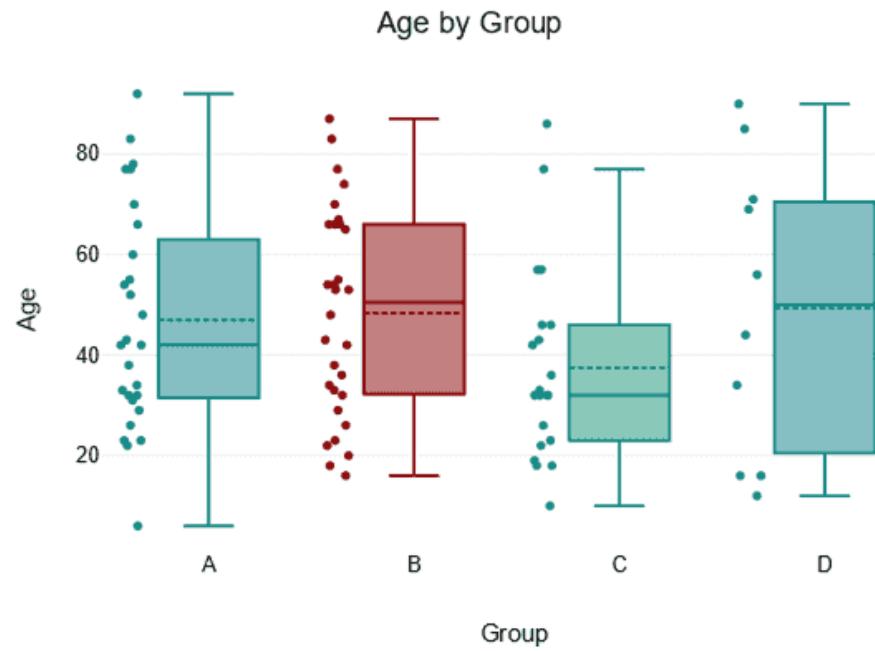
```
import seaborn as sns
import matplotlib.pyplot as plt

# Load a sample dataset
tips = sns.load_dataset("tips")

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

Output: A boxplot showing total bills for each day.

Example of Boxplot



Summary



- Data analysis helps in decision-making and discovering trends.



- Four main types: descriptive, diagnostic, predictive, prescriptive.



- Process involves multiple stages from data collection to visualization.



- Understanding data formats and Python libraries is key.

Week 2: Data Manipulation with Pandas

What is Pandas?

- Powerful Python library for data manipulation and analysis.
- Efficient and flexible data structures.
- Essential for data cleaning, transformation, and exploration.
- Builds upon NumPy, adding labeled data structures.

(Example): "Imagine you have sales data in a CSV. Pandas can read it, calculate totals, and visualize results quickly."

What is Data Manipulation?

- Transforming and organizing data for analysis.
- Examples:
 - Cleaning: Fixing typos, handling missing values.
 - Filtering: Selecting specific data based on criteria.
 - Sorting: Arranging data in order.
 - Merging: Combining data from different sources.
 - Aggregating: Summarizing data (e.g., calculating totals, averages).

Example: A Short Table Before Data Manipulation

(Do you see any problem in this data table?)

Customer ID	Name	Purchase Amount	Date	City	Discount
101	John	500	2024-01-10	New York	10%
102	Alice	NaN	2024-02-15	NaN	5%
103	Bob	700	2024-03-20	Chicago	15%
104	David	-200	2024-04-12	Houston	NaN
105	Emily	300	2024-05-05	Los Angeles	10%

Example: Data Manipulation on the table (You do not need to perform it at this stage, this is just for illustration)

Data Manipulation Steps:

- 1.Fill missing values** (fillna()): Replace missing Purchase Amount with **average value** and missing City with "**Unknown**".
- 2.Correct invalid data** (abs()): Convert negative Purchase Amount to positive.
- 3.Convert Discount to numeric values** (str.replace('%', '') & .astype(float))
- 4.Add a new column** (Total Price): Purchase Amount after applying Discount.

After Manipulation and Cleaning

Customer ID	Name	Purchase Amount	Date	City	Discount (%)	Total Price	Notes
101	John	500	2024-01-10	New York	10.0	450.0	<input checked="" type="checkbox"/> Missing Purchase Amount filled with 550 (average value)
102	Alice	550	2024-02-15	Unknown	5.0	522.5	<input checked="" type="checkbox"/> Missing City replaced with "Unknown"
103	Bob	700	2024-03-20	Chicago	15.0	595.0	<input checked="" type="checkbox"/> Negative Purchase Amount corrected to positive
104	David	200	2024-04-12	Houston	0.0	200.0	<input checked="" type="checkbox"/> Discount converted to numeric values
105	Emily	300	2024-05-05	Los Angeles	10.0	270.0	<input checked="" type="checkbox"/> Total Price calculated as Purchase Amount - Discount

Data Manipulation Capabilities of different programmes

Feature	Excel	SQL	Python (Pandas)
Data Import/Export	Supports CSV, JSON, etc.	Limited (depends on DBMS)	Supports CSV, JSON, Excel, SQL, etc.
Data Cleaning	Manual or formula-based	Limited (requires queries)	Advanced (e.g., dropna(), fillna())
Filtering	Basic (filter menus)	Advanced (WHERE, HAVING clauses)	Advanced (boolean indexing, query())
Sorting	Basic (sort menus)	Advanced (ORDER BY)	Advanced (sort_values())
Aggregation	Basic (PivotTables)	Advanced (GROUP BY, aggregate funcs)	Advanced (groupby(), agg())
Handling Missing Data	Manual or formula-based	Limited (requires queries)	Advanced (dropna(), fillna())
Data Transformation	Limited (formulas, Power Query)	Advanced (JOINS, subqueries)	Advanced (merge(), pivot_table())
Visualization	Built-in charts	Limited (requires external tools)	Advanced (Matplotlib, Seaborn)

When to Use Which Tool?

Use Case	Best Tool	Reason
Small datasets with quick analysis	Excel	Easy to use and no setup required.
Large datasets in relational databases	SQL	Efficient querying and scalability.
Data cleaning and transformation	Python (Pandas)	Advanced manipulation capabilities and flexibility.
Automation and reproducibility	Python (Pandas)	Scriptable and integrates with other tools.
Visualization and reporting	Excel or Python	Excel for quick charts; Python for advanced/custom visualizations.

Focusing on Pandas: What is a Pandas Series?

- One-dimensional labeled array (Series)
- Creating a Series (pd.Series)
- A Pandas Series is a one-dimensional labeled array that can hold data of any type (integers, strings, floats, etc.). It is similar to a column in a spreadsheet or a list in Python, but with built-in indexing.
- Operations on Series (arithmetic, statistical methods)
- (Please see slide notes for more details)

Two Components of series

- A **Pandas Series** is a one-dimensional array-like object that can hold any data type.
- It consists of two main components:
 1. **Data:** The actual values (e.g., [100, 200, 300, 400]).
 2. **Index:** Labels for the data (e.g., ['A', 'B', 'C', 'D']).

Example of a Pandas Series

Input

- import pandas as pd
- # Creating a Series with custom index
- data = pd.Series([100, 200, 300, 400], index=['A', 'B', 'C', 'D'])
- # Displaying the Series
- print(data)

Output:

- A 100
- B 200
- C 300
- D 400

Each value is associated with a corresponding index label.

The index allows you to access specific values using their labels

What is a Pandas DataFrame?

- Two-dimensional labeled data structure
- Understanding rows and columns
- Operations on a DataFrame (Please see slide notes)

Example of Data Frame

```
import pandas as pd
data = {'Name': ['Alice', 'Bob'], 'Score': [85, 90]}
df = pd.DataFrame(data)
print(df)
```

- Explanation of how DataFrames work
- Different ways to create a DataFrame (from lists, dictionaries, CSV files, etc.)

Visual Representation of a Dataframe

Index	Name	Age	City	Salary
0	Alice	25	New York	\$50,000
1	Bob	30	Chicago	\$55,000
2	Charlie	35	Los Angeles	\$60,000
3	David	40	Houston	\$65,000

Indexing in Pandas

- **Indexing allows you to access and manipulate data in a Series using labels or positions.**
- Similar to lists and dictionaries in Python, but with more flexibility.
- Two main types of indexing:
 1. **Label-based indexing (.loc[])** – Custom Indexing
 2. **Position-based indexing (.iloc[])** – Default Indexing

Default Indexing in Pandas

- **The default index is a numeric sequence (0, 1, 2, ...) assigned to rows or elements when no custom index is provided.**
- **It provides a unique identifier for each row or element in the Series or DataFrame.**

Default Indexing Example

```
import pandas as pd  
  
0    100      # Creating a Series without  
1    200      specifying an index  
2    300      data = pd.Series([100, 200, 300,  
3    400      400])  
              # Displaying the Series  
print(data)
```

Creating a Pandas Series with Custom Index

A	100
B	200
C	300
D	400

```
import pandas as pd  
  
# Creating a Series with custom index  
data = pd.Series([100, 200, 300, 400],  
index=['A', 'B', 'C', 'D'])  
  
# Displaying the Series  
print(data)
```

Accessing Data Using Index

Code

```
# Accessing the value for index 'B'  
print(data['B']) # Output: 200
```

Index	Data
A →	100
B →	200 ← Accessed using data['B']
C →	300
D →	400

Slicing Data Using Index

- # Slicing the Series from index 'B' to 'D'
- print(data['B':'D'])

Index	Data	
A →	100	
B →	200	– Start of slice
C →	300	
D →	400	– End of slice

B	200
C	300
D	400

Example - Indexing and Slicing Data

- `df.loc[0]` # Accessing first row
- `df.iloc[:, 1]` # Accessing second column
- `df[df['Score'] > 85]` # Filtering rows where Score > 85

Indexing and accessing in dataframe (instead of series)

1. Why Access Rows and Columns Together?

- To extract specific data points or subsets of data from a DataFrame.
- Useful for filtering, analyzing, or manipulating data.

2. Methods for Accessing Rows and Columns:

- **loc[]**: Access data using **row and column labels**.
- **iloc[]**: Access data using **row and column positions**.

Label-Based Indexing

Basic Syntax : df.loc[row_label, column_label]

Example

```
import pandas as pd
# Creating a DataFrame
data = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [25, 30, 35, 40],
    'City': ['New York', 'Chicago', 'Los Angeles', 'Houston'],
    'Salary': ['$50,000', '$55,000', '$60,000', '$65,000']
}, index=['A', 'B', 'C', 'D'])
# Accessing a specific row and column using labels
print(data.loc['B', 'Age']) # Output: 30
```

Accessing multiple rows and columns

- # Accessing multiple rows and columns
- `print(data.loc[['A', 'C'], ['Name', 'Salary']])`

	Name	Salary
A	Alice	\$50,000
C	Charlie	\$60,000

Filtering Data in a DataFrame

1. What is Filtering?

1. Filtering allows you to extract rows from a DataFrame that meet specific conditions.
2. It is useful for analyzing subsets of data.

- Syntax
- `df[condition]`
- **Methods for Filtering:**
- **Boolean Indexing:** Use conditions to create a Boolean mask and filter rows.
- **query() Method:** Filter rows using a query string.

Filtering using Boolean Indexing

```
import pandas as pd

# Creating a DataFrame
data = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [25, 30, 35, 40],
    'City': ['New York', 'Chicago', 'Los Angeles', 'Houston'],
    'Salary': ['$50,000', '$55,000', '$60,000', '$65,000'] })
# Filtering rows where Age is greater than 30
filtered_data = data[data['Age'] > 30]
print(filtered_data)
```

Filtering - Example

```
import pandas as pd

# Creating a DataFrame

data = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [25, 30, 35, 40],
    'City': ['New York', 'Chicago', 'Los Angeles', 'Houston'],
    'Salary': ['$50,000', '$55,000', '$60,000', '$65,000'] })

# Filtering rows where Age is greater than 30

filtered_data = data[data['Age'] > 30]

print(filtered_data)
```

Filtering - Output

	Name	Age	City	Salary
2	Charlie	35	Los Angeles	\$60,000
3	David	40	Houston	\$65,000

Filtering Using query() Method

- **Syntax:**

```
df.query('condition')
```

- **Example:**

```
# Filtering rows where Age is greater than 30 using query()  
filtered_data = data.query('Age > 30')  
print(filtered_data)
```

Output – Query based filtering

	Name	Age	City	Salary
2	Charlie	35	Los Angeles	\$60,000
3	David	40	Houston	\$65,000

Key Points to Remember

- Use **Boolean indexing** for simple and complex conditions.
- Use the **query()** method for concise and readable filtering.
- Filtering returns a new DataFrame with rows that meet the condition(s).

Introduction to Missing Data

- - Missing data occurs when some values in a dataset are unavailable.
- - Common reasons: data entry errors, survey dropouts, system failures.
- - Poor handling can lead to biased analysis and inaccurate results.

Handling missing data

- **Values considered “missing”**
- pandas uses different sentinel values to represent a missing (also referred to as NA) depending on the data type.
-

Missing Values in Different Data Types

Data Type	Sentinel Value for Missing Data
Floats	NaN (Not a Number, from NumPy)
Strings	None or NaN
Boolean	NaN (Converts column to object type)
Datetime	NaT (Not a Time)

Scenario 1: Survey Data (Incomplete Responses)

- - People skip questions due to privacy concerns or interruptions.
- - Example: Missing age, income, or satisfaction ratings in a customer survey.
- - Handling in Python:
 - - Drop missing values.
 - - Fill with mean/median/mode.

Python Example: Handling Missing Survey Data

```
import pandas as pd  
survey_data = pd.DataFrame({  
    'Age': [25, 30, None, 45, 29],  
    'Income': [50000, None, 60000, 75000, None]})  
print(survey_data.isnull().sum())
```

Scenario 2: Financial Data (Missing Stock Prices)

- - Missing stock prices due to market holidays or data feed failures.
- - Can impact trend analysis and investment decisions.
- - Handling in Python:
 - - Forward-fill to propagate last known value.
 - - Interpolation for trend-based missing values.

Python Example: Handling Missing Financial Data

```
import pandas as pd  
stock_prices = pd.DataFrame({  
'Date': pd.date_range('2024-02-10', periods=5, freq='D'),  
'Stock_Price': [150, None, 152, None, 155]})  
stock_prices.fillna(method='ffill', inplace=True)
```

Scenario 3: Healthcare Data (Incomplete Patient Records)

- - Missing patient test results due to skipped checkups or entry errors.
- - Missing data can affect medical diagnoses and ML models.
- - Handling in Python:
 - - Use KNN Imputation to estimate missing values. (**More details on KNN will be covered in the Machine Learning Course**)
 - - Drop if too many values are missing.

Python Example: Handling Missing Healthcare Data

```
from sklearn.impute import KNNImputer  
import pandas as pd  
  
health_data = pd.DataFrame({  
    'Blood_Pressure': [120, 130, None, 110, 140],  
    'Cholesterol': [200, None, 220, 210, None]})  
  
imputer = KNNImputer(n_neighbors=2)  
  
health_data_imputed =  
pd.DataFrame(imputer.fit_transform(health_data))
```

Summary: Handling Missing Data

- - **Survey Data:** Fill missing values using mean/median.
- - **Financial Data:** Use forward-fill or interpolation.
- - **Healthcare Data:** Use KNN imputation or drop columns with too many missing values.
- - Always analyze missing data before choosing a method.

Examples (Reindexing)

- In [1]: `pd.Series([1, 2], dtype=np.int64).reindex([0, 1, 2])`
- Out[1]:
 - 0 1.0
 - 1 2.0
 - 2 NaN
- dtype: float64

- In [2]: `pd.Series([True, False], dtype=np.bool_).reindex([0, 1, 2])`
- Out[2]:
 - 0 True
 - 1 False
 - 2 NaN
- dtype: object

Detecting missing values

- `ser = pd.Series([pd.Timestamp("2020-01-01"), pd.NaT])`

`ser`

0 2020-01-01

1 NaT

- `dtype: datetime64[ns]`

- `pd.isna(ser)`

- 0 False

- 1 True

- `dtype: bool`

isna() or notna() will also consider None a missing value

- isna() is a function in Pandas that checks for missing values (NaN or None) in a DataFrame or Series and returns a Boolean mask (True for missing values, False for non-missing values).
- `ser = pd.Series([1, None], dtype=object)`

```
0      1  
1    None  
dtype: object
```

```
0    False  
1    True  
dtype: bool
```

Calculations with missing data

- Missing values propagate through arithmetic operations between pandas objects
- When summing data, NA values or empty data will be treated as zero.

- ser1 = pd.Series([np.nan, np.nan, 2, 3])
 - ser2 = pd.Series([np.nan, 1, np.nan, 4])
 - ser1
 - 0 NaN
 - 1 NaN
 - 2 2.0
 - 3 3.0
 - dtype: float64
 - ser2
 - 0 NaN
 - 1 1.0
 - 2 NaN
 - 3 4.0
 - float64
-
- ser1 + ser2
 - 0 NaN
 - 1 NaN
 - 2 NaN
 - 3 7.0
 - dtype: float64

Summing data with missing values

- In [75]: `pd.Series([np.nan]).sum()`
- Out[75]: 0.0

- In [76]: `pd.Series([], dtype="float64").sum()`
- Out[76]: 0.0

Product with missing values

- When taking the product, NA values or empty data will be treated as 1.
- `pd.Series([np.nan]).prod()`
- `1.0`
- `pd.Series([], dtype="float64").prod()`
- `1.0`

Dropping missing data

```
df = pd.DataFrame([[np.nan, 1, 2], [1, 2, np.nan], [1, 2, 3]])
```

```
df
```

	0	1	2
0	NaN	1	2.0
1	1.0	2	NaN
2	1.0	2	3.0

```
df.dropna()
```

Column Drop

	1
df.dropna(axis=1)	0 1
	1 2
	2 2

Default Axis

Axis 0 = Rows

Axix 1 = Column

`dropna()` (default `axis=0`) removes rows containing `NaN`.

`dropna(axis=1)` removes columns containing `NaN`.

If you want to drop rows or columns only when all values are `NaN`, use:

```
df.dropna(how="all")
```

To fill missing values instead of dropping them, use:

```
df.fillna(0) # Replace NaN with 0
```

Key Takeaways

1. **None is treated as missing in Pandas, just like NaN.**
2. **isna() identifies missing values (None or NaN) and returns True.**
3. **Works with object dtype as well, not just numeric types.**
4. **Use notna() if you want to find non-missing values**

Week 3

Data Cleaning and Manipulation
(Duplicates, Outliers)

Week 3 Objectives

- This week, we will reinforce last week's learning with some advanced analyses that includes:
 - Discussing advanced techniques for data cleaning including handling duplicate data
 - Understanding the adverse impacts of uncleaned data
 - Analyzing Outliers and fixing outliers' issues
 - Groupby Analysis
 - Organizing data with Pivot tables
 - Organizing data with Cross-Tabulation

Issues Arising Due to Duplicates in Data

- Duplicates in a dataset can lead to **incorrect analysis, biased insights, and operational inefficiencies**. Below are some common problems caused by duplicate data in different domains.
 - Inaccurate Statistical Analysis
 - If the same data point appears multiple times, it skews statistical results and misrepresents trends.
 - Example: Sales Analysis A retail company analyzing monthly sales finds duplicate entries in sales data

Fixing Duplicates

```
import pandas as pd
```

```
df = pd.DataFrame({  
    'CustomerID': [101, 102, 102, 103, 104, 104],  
    'TotalPurchase': [200, 150, 150, 300, 400, 400]})
```

- `print(df['TotalPurchase'].mean()) # Incorrect if duplicates exist`

Misleading Machine Learning Models

- Duplicate data can cause **bias** in model training, making **overfitting more likely**.
- **Example: Fraud Detection Model**
If fraudulent transactions are duplicated, the model might **learn to classify them more often than necessary**, making it **less reliable on new data**.

Redundant Storage and Increased Costs

- Large datasets with duplicate records take up **more storage space**, slowing down processing.
- **Example: Cloud Storage Costs**
If a company's **customer database** contains duplicate records, unnecessary storage costs accumulate over time.

Incorrect Business Decisions

Executives rely on **data-driven decisions**, but duplicates can **mislead them** into taking incorrect actions.

Example: Inventory Management
A warehouse system mistakenly shows **double the available stock** of a product due to duplicated inventory records.
Consequence: The business **under-orders inventory**, leading to stock shortages.

Skewed Financial Reports

- Financial teams might **double count revenues, expenses, or liabilities**, resulting in **misreported earnings**.
- **Example: Banking Transactions**
A bank processes a duplicate transaction entry, leading to **incorrect account balances** for customers.

Customer Dissatisfaction & Poor User Experience

- Customers receive the same promotional email multiple times, leading to frustration and increased unsubscribe rates.
- Example: Duplicate Emails in Marketing



Compliance and Legal Issues

- Many industries (finance, healthcare) require **accurate and unique records** for compliance.
- **Example: Healthcare Records**
Duplicate patient records lead to **incorrect medical history tracking**, which can result in **serious legal and medical consequences**.

Removing Duplicate Rows (`drop_duplicates()`)

- import pandas as pd
- df = pd.DataFrame({
- 'ID': [1, 2, 2, 3, 4, 4, 5],
- 'Name': ['Alice', 'Bob', 'Bob', 'Charlie', 'David', 'David', 'Eve'],
- 'Age': [25, 30, 30, 35, 40, 40, 22]
- })
- print(df)
- print("Before removing duplicates:")print(df)
- df_unique = df.drop_duplicates()
- print("\nAfter removing duplicates:")
- print(df_unique)

- Before removing duplicates:

	ID	Name	Age
0	1	Alice	25
1	2	Bob	30
2	2	Bob	30
3	3	Charlie	35
4	4	David	40
5	4	David	40
6	5	Eve	22

- After removing duplicates:

	ID	Name	Age
0	1	Alice	25
1	2	Bob	30
3	3	Charlie	35
4	4	David	40
6	5	Eve	22

Removing Duplicates from Specific Columns

- If you want to remove duplicates based on a specific column, use:
- `df_unique = df.drop_duplicates(subset=['Name'])`

Keeping the Last Occurrence

- By default, `drop_duplicates()` keeps the first occurrence. To keep the last one instead:
- `df_unique = df.drop_duplicates(keep='last')`

Example

- Default behaviour is keeping the first occurrence and dropping the rest

	ID	Value	
0	A	10	
1	B	20	
2	A	15	# Duplicate of row 0 (same ID)
3	C	30	
4	A	25	# Another duplicate of row 0 and 2

- But if you use `keep=last`, then it would retain the 5th row and drop 0 and 2 rows

	ID	Value
1	B	20
3	C	30
4	A	25

Removing All Duplicate Entries

- To **remove all duplicate rows**, use:
- `df_no_dupes = df.drop_duplicates(keep=False)`
-

	ID	Value
1	B	20
3	C	30

Removing Duplicates and Updating the Original DataFrame

- If you want to remove duplicates in place, without creating a new DataFrame:
- `df.drop_duplicates(inplace=True)`

Checking for Duplicates Before Removal

- Before removing duplicates, check how many exist:
- `print(df.duplicated().sum())` # Count duplicate rows
- `print(df.duplicated(subset=['Name']).sum())` # Count duplicates in 'Name'

Outliers in Data & Issues in Business Analytics

- **What Are Outliers?**
- Outliers are **data points that are significantly different** from the rest of the dataset. They can be **too high or too low** compared to the majority of values and may arise due to **measurement errors, fraud, or rare events**.
- Outliers can **skew business analytics, mislead decision-making, and distort machine learning models**.

Why Do Outliers Occur?

Cause	Example
Data Entry Errors	A salesperson accidentally enters \$100,000 instead of \$1,000 .
Fraudulent Activity	A hacker makes unusually large transactions on a stolen credit card.
Market Anomalies	A stock price jumps 200% in a day due to a sudden announcement.
Operational Errors	A temperature sensor malfunctions , recording -200°C .
Genuine Rare Events	A business gets an unexpected billion-dollar order .

Issues Caused by Outliers in Business Analytics

A) Skewed Financial Reports

- **Example:** A company's average monthly sales report is inflated due to **one unusually high sale**.
- **Impact:** Misleading financial decisions, incorrect revenue projections.
- **Solution:** Use **median** instead of **mean** for robust metrics.

Issues Caused by Outliers in Business Analytics

- **B) Misleading Customer Insights**
- **Example:** A supermarket assumes the **average spending per customer** is \$2,000, but one VIP customer who spent \$100,000 skews the data.
- **Impact:** Marketing campaigns **target the wrong customer base**.
- **Solution:** Remove extreme outliers before calculating insights.

Issues Caused by Outliers in Business Analytics

- **C) Incorrect Inventory Planning**
- **Example:** An online store sees a **sudden surge in demand** due to a viral trend and **over-orders inventory**, but the demand normalizes.
- **Impact:** Excess stock, **wasted storage costs**, and financial losses.
- **Solution:** Use **historical trends** and **remove temporary anomalies** from forecasting models.

Issues Caused by Outliers in Business Analytics

D) Poor Machine Learning Model Performance

- Example: A fraud detection model is trained with data containing extreme outliers, causing it to flag legitimate transactions as fraud.
- Impact: False positives, leading to customer dissatisfaction.
- Solution: Use robust scaling methods (e.g., log transformation, IQR filtering).

Business Scenarios Where Outlier Detection is Crucial

Industry	Problem Due to Outliers	Solution
Retail & E-Commerce	Skewed average customer spending due to VIP buyers	Use median spending, remove extreme outliers
Finance & Banking	Fraudulent transactions mistaken for real spending	Use machine learning-based anomaly detection
Healthcare	Faulty medical device readings affecting diagnosis	Filter sensor data using IQR method
Stock Market	One-day stock spikes distorting trend analysis	Use rolling averages to smooth trends
Manufacturing	Defective products causing false alarms	Implement Z-score filtering

Handling Outliers in Pandas

- Outliers are **extreme values** that deviate significantly from other data points. Handling them properly is crucial for **accurate business analytics, machine learning, and financial modeling**.
- **Using IQR (Interquartile Range) Method**
 - IQR measures the **spread of the middle 50% of data** and identifies extreme values.

A) Using Boxplots (Visual Detection)

- import pandas as pd
- import matplotlib.pyplot as plt
- # Sample data with outliers
- df = pd.DataFrame({'Sales': [1000, 1200, 1300, 1500, 2000, 50000]})
- # 50,000 is an outlier
- # Create a boxplot
- plt.boxplot(df['Sales'])plt.title("Boxplot of Sales")plt.show()
Outliers appear as points outside the whiskers.

B) Using Z-Score Method

Z-score measures how far a data point is from the mean in terms of standard deviations.

```
from scipy.stats import zscore
```

```
df['Z_Score'] = zscore(df['Sales']) # Compute Z-score
```

```
outliers = df[df['Z_Score'].abs() > 3] # Outliers are those with |Z| > 3
```

```
print(outliers) # Displays only the extreme values
```

C) IQR filtering

- `Q1 = df['Sales'].quantile(0.25) # 25th percentile`
- `Q3 = df['Sales'].quantile(0.75) # 75th percentile`
- `IQR = Q3 - Q1 # Compute IQR`
- `# Define outlier boundaries`
- `lower_bound = Q1 - 1.5 * IQR`
- `upper_bound = Q3 + 1.5 * IQR`
- `outliers = df[(df['Sales'] < lower_bound) | (df['Sales'] > upper_bound)]`
- `print(outliers) # Displays outlier values`

Explanation

Index	Sales	
0	1000	
1	1200	 25 th Percentile
2	1300	
3	1500	
4	2000	 75 th Percentile
5	50000	

$$\text{Lower Bound} = Q1 - 1.5 \times \text{IQR} = 1200 - (1.5 \times 800) = 1200 - 1200 = 0$$

$$\text{Upper Bound} = Q3 + 1.5 \times \text{IQR} = 2000 + (1.5 \times 800) = 2000 + 1200 = 3200$$

Since 50,000 is outside the upper bound, it is an outlier

Handling Outliers in Pandas

- A) Removing Outliers
- If outliers are caused by data entry errors, removing them is the best approach.
- # Remove outliers using IQR filtering
- ```
df_cleaned = df[(df['Sales'] >= lower_bound) & (df['Sales'] <= upper_bound)]
```
- ```
print(df_cleaned)
```

 # Data without outliers
- Removes extreme values while keeping normal data.

Handling outliers

- B) Replacing Outliers (Capping)
- If outliers contain valuable information, we can replace them instead of removing them.
 - (i) Capping Using IQR Limits
 - `df['Sales'] = df['Sales'].clip(lower_bound, upper_bound) # Caps extreme values`
 - All outliers are replaced with the nearest valid values.

Handling outliers

- (ii) Replacing Outliers with Mean or Median
- `median_value = df['Sales'].median() # Compute median`
- `df.loc[(df['Sales'] < lower_bound) | (df['Sales'] > upper_bound), 'Sales'] = median_value # Replace outliers`
- Useful when outliers may still provide business value.

Handling outliers

- **C) Transforming Data**
- Instead of removing outliers, we can **apply mathematical transformations** to reduce their impact.
- **(i) Log Transformation (Reduces Spread)**
- import numpy as np
- `df['Sales_Log'] = np.log1p(df['Sales'])`

Handling outliers

- (ii) Winsorization (Replaces Extreme Values)
- from scipy.stats.mstats import winsorize
- df['Sales_Winsorized'] = winsorize(df['Sales'], limits=[0.05, 0.05]) # Caps 5% extreme values
- Keeps all data but limits extreme influence.

Week 4

Data Cleaning and Manipulation

(Groupby , Pivot Tables, Common Errors and Fixes)

Introduction to GroupBy

- GroupBy is a powerful feature in Pandas that allows you to split data into groups based on one or more columns, apply operations to each group independently, and then combine the results. It follows the "split-apply-combine" strategy.
- Example: Grouping students by grade level and then calculating the average score for each grade.
- Example: Finding average house price per suburb
- Syntax : `df.groupby("Suburb")["Price"].mean()`

Comparison of Groupby in Python and Excel

- **Comparison with Excel:**
 - . `df.groupby("Suburb")["Price"].mean()` → Similar to **Excel Pivot Table's Average Function**
 - . `df.groupby(["Suburb", "Type"])["Price"].sum()` → Similar to **SUMIFS in Excel**

Before Groupby

Suburb	Type	Rooms	Price
Abbotsford	h	3	1200000
Abbotsford	h	2	1000000
Alphington	h	4	1800000
Alphington	u	2	700000
Balwyn	h	3	1500000
Balwyn	t	2	900000
Brighton	h	5	2500000
Brighton	u	1	500000
Carlton	u	2	600000
Carlton	t	3	1100000

After Groupby “Type”

Type	Average Price
h	1500000
t	1000000
u	600000

Explanation:

- 1. Split:** The data is split into three groups based on the unique values in the "Type" column: 'h', 't', and 'u'.
- 2. Apply:** The mean() function (average) is applied to the "Price" column within each group.
- 3. Combine:** The results (the average prices) are combined into a new table, where the "Type" becomes the index (or group label), and the "Average Price" is the calculated value.

Aggregation Functions

Aggregation functions summarize numerical data into single values.

Common functions:

- `sum()` – Total sum of values
- `mean()` – Average value
- `count()` – Number of non-null values
- `min()` / `max()` – Minimum & Maximum values
- `median()` – Middle value
- `std()` – Standard deviation

Using Aggregation Function with other common functions

```
df.groupby("Suburb")["Price"].agg(["mean", "min", "max", "count"])
```

Output of groupby() with Aggregation:

Suburb	Mean Price	Min Price	Max Price	Count
Carlton	745000	730000	760000	2
Fitzroy	990000	970000	1020000	3
Richmond	881667	850000	920000	3

Using Multiple GroupBy Keys

- We can group by multiple columns for deeper insights.
- Example: Grouping by Suburb & Rooms
- Helps in multi-level analysis of grouped data.
 - Suburb (e.g., Richmond, Carlton)
 - Property Type (e.g., House, Apartment)
 - Rooms (e.g., 2, 3, 4)
 - Price (Property price)
- `df.groupby(["Suburb", "Rooms"])["Price"].mean()`

Multi-Level GroupBy Table (Mean Price by Suburb & Property Type)

Suburb	Property Type	Average Price (\$)
Carlton	Apartment	600,000
	House	850,000
Fitzroy	House	1,090,000
Richmond	Apartment	650,000
	House	1,000,000

Introduction to Pivot Tables

- What is a Pivot Table?
 - A tool for dynamic summarization of data.
 - Aggregates data based on different categories.
 - Useful in data reporting & business intelligence.
 - Example: Finding the average price per suburb

```
df.pivot_table(index="Suburb", values="Price",  
aggfunc="mean")
```

GroupBy vs. Pivot Tables in Pandas

Both Groupby and Pivot give similar output (not exactly same) but use different approaches

```
df.groupby(["Suburb", "Type"])["Price"].mean()
```

```
df.pivot_table(index="Suburb", columns="Type",  
values="Price", aggfunc="mean")
```

- **Groupby Output** (creates a long-format table, great for numerical analysis.)

Suburb	Type	Avg Price
Carlton	Apartment	600,000
Carlton	House	850,000
Fitzroy	House	1,090,000
Richmond	Apartment	650,000
Richmond	House	1,000,000

- **Pivot Output** (creates a wider, spreadsheet-like table, useful for visualization & reports.)

Suburb	Apartment	House
Carlton	600,000	850,000
Fitzroy	Nan	1,090,000
Richmond	650,000	1,000,000

Handling Missing Values in Pivot Tables

- Pivot tables can handle missing data using `fill_value`.
- Example: Replacing missing values with 0
`df.pivot_table(index="Suburb",
values="Price", aggfunc="mean",
fill_value=0)`

Introduction to Cross-Tabulation

- pd.crosstab() computes frequency tables across categories.
- Helps analyze relationships between **categorical variables**.
- Example: Count the number of properties per suburb & type

Percentage-Based Cross-Tabulation of Suburb vs. Property Type

Suburb	Apartment (%)	House (%)
Carlton	50.0%	50.0%
Fitzroy	33.3%	66.7%
Richmond	33.3%	66.7%

Interpretation and Uses

- **Interpretation:**
 - Carlton has an **equal split (50% Apartments, 50% Houses)**.
 - Fitzroy & Richmond have **more Houses (66.7%) than Apartments (33.3%)**.
 - **Useful for Market Analysis:** Identifies **which suburbs have more apartments vs. houses**.
- **Why is this useful?**
 - **Market Segmentation:** Helps in planning new property developments.
 - **Business Insights:** Property agencies can **target areas based on property type demand**.
 - **Investment Decisions:** Buyers can choose locations based on housing availability.

Some Advanced Python Concepts

Pip Install

- **pip install** is a command used to download and install Python packages from the Python Package Index (PyPI) or other repositories.
- **When to use it:** You use pip install when you want to add a new library or package to your Python environment.
- **Where it runs:** It is executed in the **command line** (terminal or command prompt), not in a Python script.
 - Example : `pip install pandas` (This command installs the pandas library so you can use it in your Python programs.)
 - `!pip install pip install numpy==1.21.0` (Installs version 1.21.0 of the numpy library)

Some more pip installs

Install the Latest Version of a Package

```
pip install --upgrade pandas
```

Installing Multiple Packages

```
!pip install numpy pandas matplotlib seaborn
```

Import in Python

I

Import

- `import` is a Python keyword used to include and use a module or library in your Python script.
- When to use it: You use `import` in your Python code to access the functionality of a library or module that has already been installed.
- Where it runs: It is written inside a Python script or interactive Python session (e.g., Jupyter Notebook).
- Example: `import pandas as pd`

Google Colab comes with many popular Python libraries pre-installed (e.g., numpy, pandas, matplotlib, tensorflow, etc.), but if you need additional packages, you can install them using pip.

To check which libraries are
installed in Colab

`!pip list`

Aspect	<code>pip install</code>	<code>import</code>
Purpose	Installs a package or library from PyPI or another source.	Loads an installed package or module into your Python script.
Where it's used	Command line (terminal or command prompt).	Python script or interactive session.
When to use it	When you need to install a new library or package.	When you want to use an already installed library or module in your code.
Example	<code>pip install numpy</code>	<code>import numpy as np</code>
Dependencies	Downloads and installs the package and its dependencies.	Assumes the package is already installed.
Scope	Affects the entire Python environment (global or virtual environment).	Affects only the current script or session where it is imported.

How Pip Install and Import Work Together

1. Step 1: Install the Package

- Use `pip install` to download and install the package.
- Example: `pip install matplotlib`

2. Step 2: Import the Package

- Use `import` to include the package in your Python script.
- Example: `import matplotlib.pyplot as plt`

Common Errors and Fixes in pip

Error: ModuleNotFoundError

- **Cause:** The package is not installed in your environment.
- **Fix:** Use pip install to install the missing package.
- `pip install <package_name>`

Common Errors and Fixes in pip

Error: ImportError

- **Cause:** The package is installed, but there's an issue with the import statement (e.g., incorrect module name).
- **Fix:** Double-check the import statement and ensure the package is installed correctly.

Common Errors in Python Codes and Fixes

Why Understanding Errors is Important

- Errors help identify issues in code.
- Properly reading errors speeds up debugging.
- Python provides clear error messages to guide fixes.

SyntaxError

Cause: Invalid Python syntax.

Example: `print("Hello, World"`

Error Message: `SyntaxError: unexpected EOF while parsing`

IndentationError

Cause: Incorrect indentation.

Example:

```
def greet():
    print("Hello, World")
```

Error Message: IndentationError: expected
an indented block

Fix: Indent the code block properly.

NameError

Cause: Using an undefined variable or function.

Example:

```
print(message)
```

Error Message: NameError: name 'message' is not defined

Fix: Define the variable before use.

TypeError

Cause: Operation on incompatible types.

Example:

```
result = "5" + 3
```

Error Message: TypeError: can only
concatenate str (not "int") to str

Fix: Convert types appropriately.

IndexError

Cause: Accessing an invalid index in a sequence.

Example:

```
my_list = [1, 2, 3]
```

```
print(my_list[3])
```

Error Message:

IndexError: list index out of range

Fix: Ensure the index is within range.

```
print(my_list[-1])
```

KeyError

Cause: Accessing a non-existent dictionary key.

Example:

```
my_dict = {"name": "Alice"}  
print(my_dict["age"])
```

Error Message:

KeyError: 'age'

AttributeError

Cause: Accessing a non-existent attribute or method.

Example:

```
my_list = [1, 2, 3]
```

```
my_list.appendx(4)
```

Error Message:

```
AttributeError: 'list' object has no attribute 'appendx'
```

Fix: Use the correct attribute or method.

```
my_list.append(4)
```

ValueError

Cause: Invalid value for an operation.

Example:

```
int("abc")
```

Error Message:

```
ValueError: invalid literal for int() with base 10: 'abc'
```

Fix: Ensure the value is valid for conversion.

```
int("123")
```

ImportError

Cause: Unable to import a module.

Example:

```
import non_existent_module
```

Error Message:

```
ImportError: No module named 'non_existent_module'
```

Fix: Install or check the module name.

```
pip install module_name
```

ZeroDivisionError

Cause: Division by zero.

Example:

```
result = 10 / 0
```

Error Message:

```
ZeroDivisionError: division by zero
```

Fix: Check for zero before dividing.

```
if denominator != 0:
```

```
    result = 10 / denominator
```

Key Take aways:

- . Errors help in debugging and improving code.
- . Understanding error messages is crucial for efficient troubleshooting.
- . Regular practice makes debugging easier and faster.

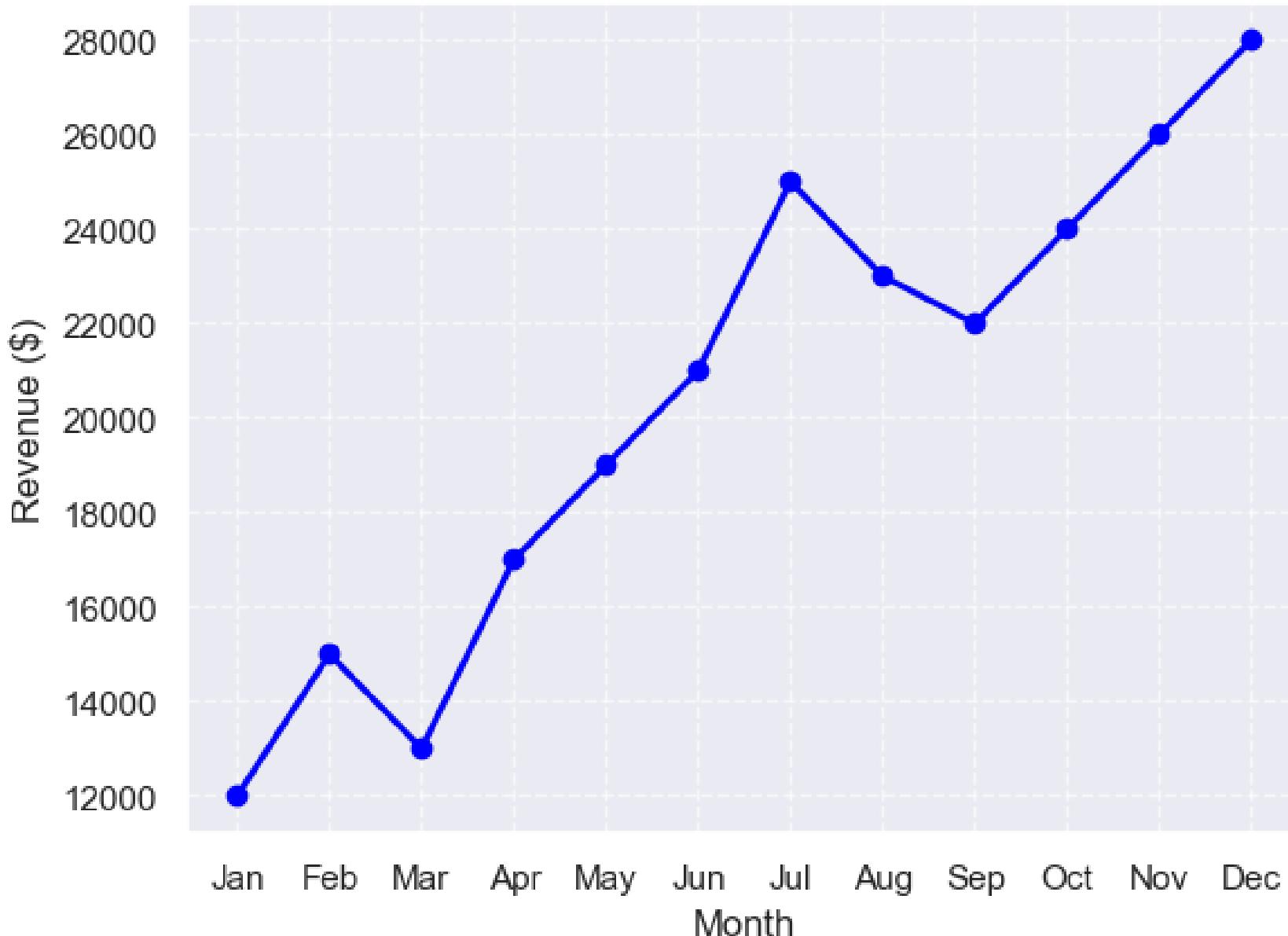
Week 5

BDA with Python

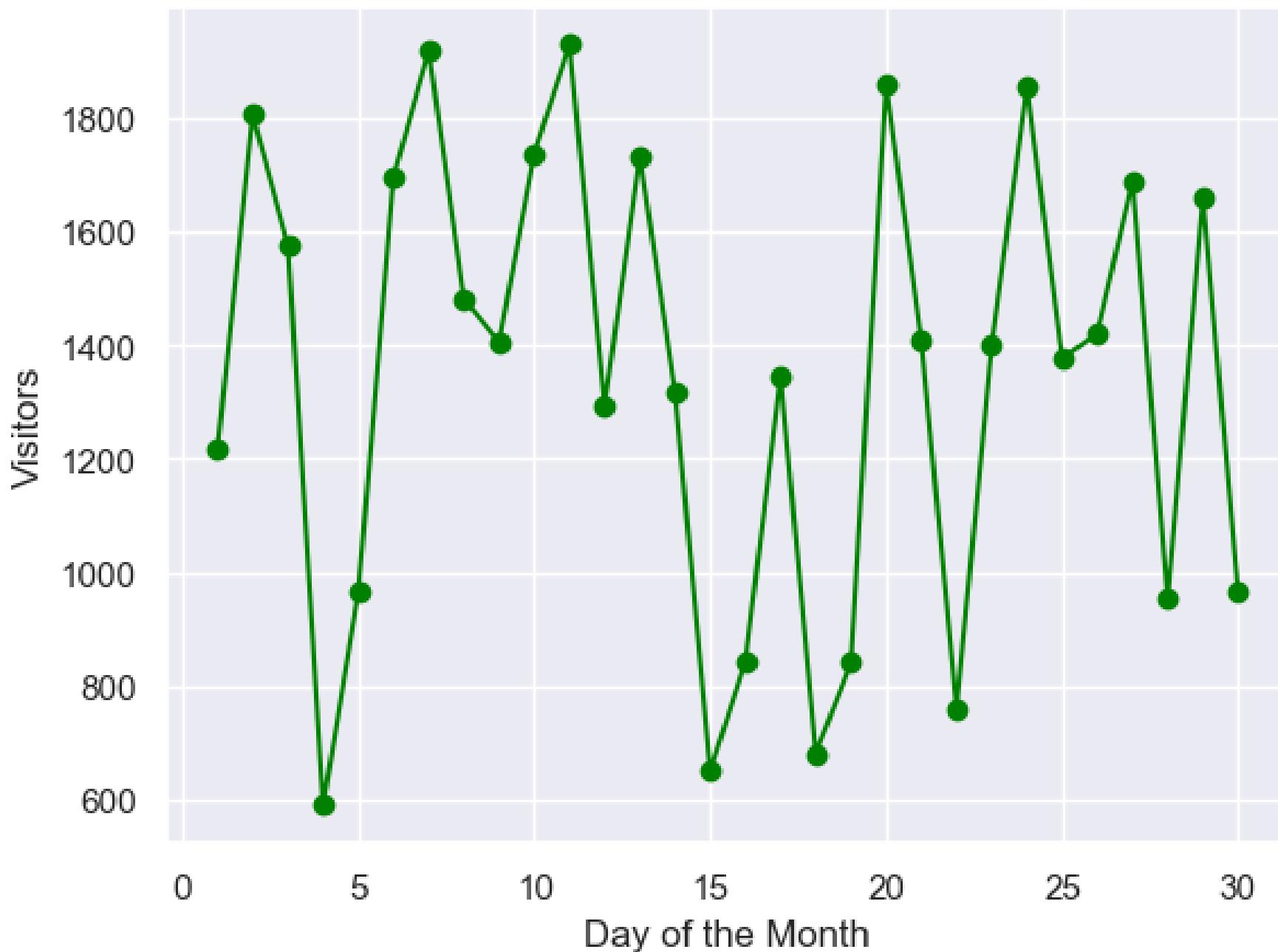
Introduction to Matplotlib - basic plots

- **Matplotlib** is a popular Python library for creating static, animated, and interactive visualizations. It provides an interface similar to **MATLAB** and works well with **NumPy** and **pandas**.
- Installing & Importing MatplotlibBefore using Matplotlib, install it (if not already installed):
 - install matplotlib
- Then, import the necessary modules:
 - import matplotlib.pyplot as plt
 - import numpy as np

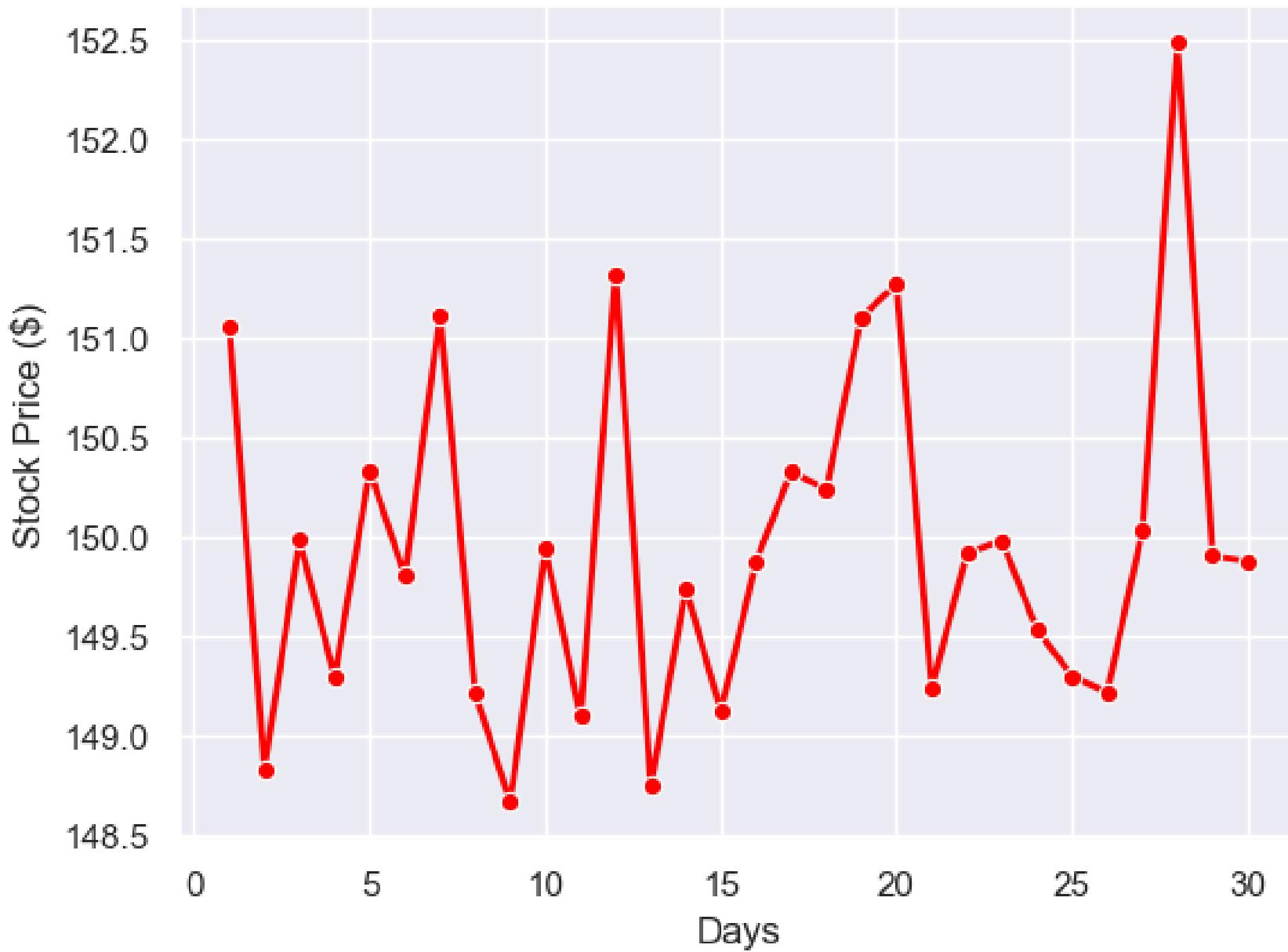
Monthly Sales Trend



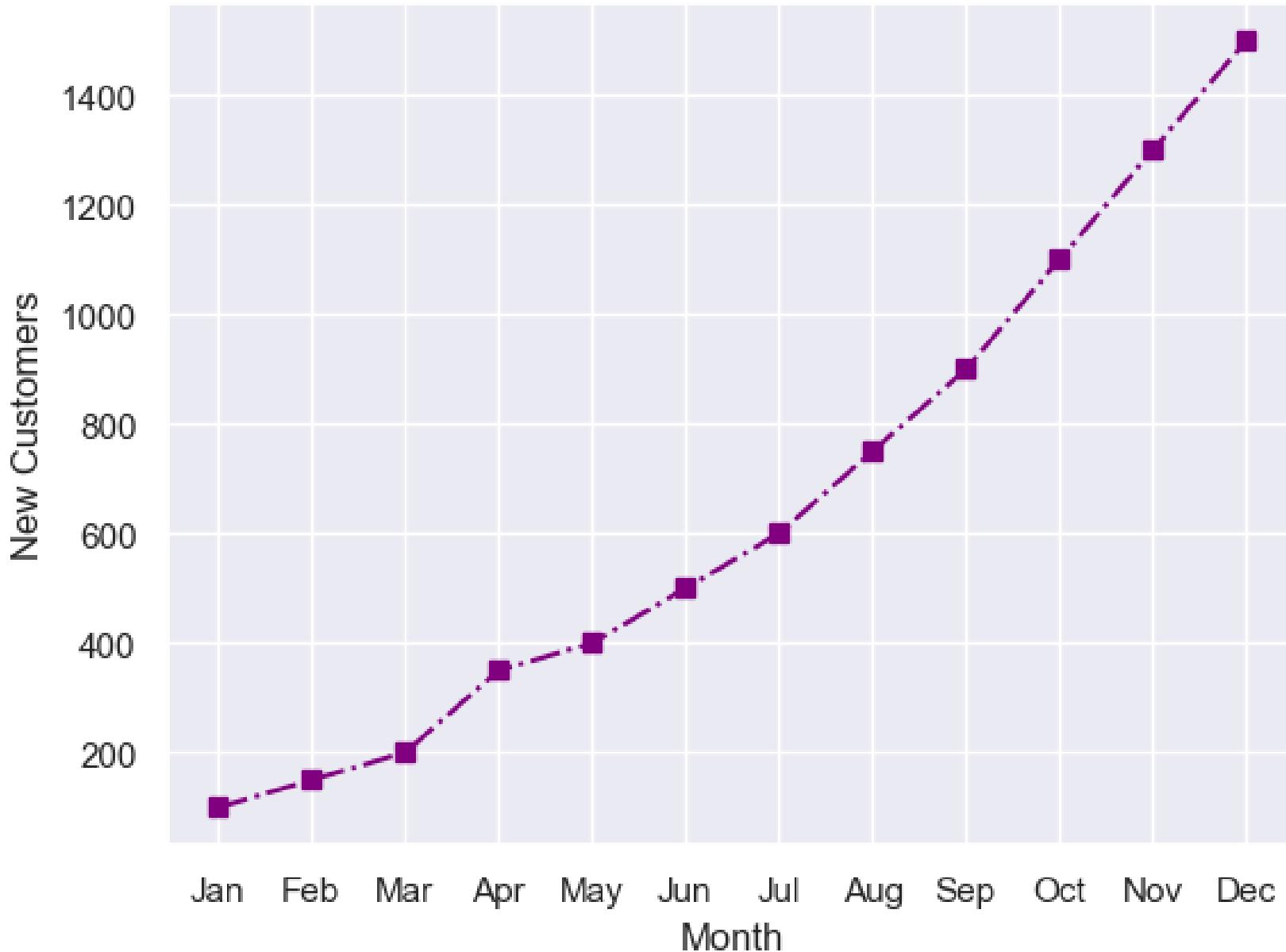
Daily Website Traffic



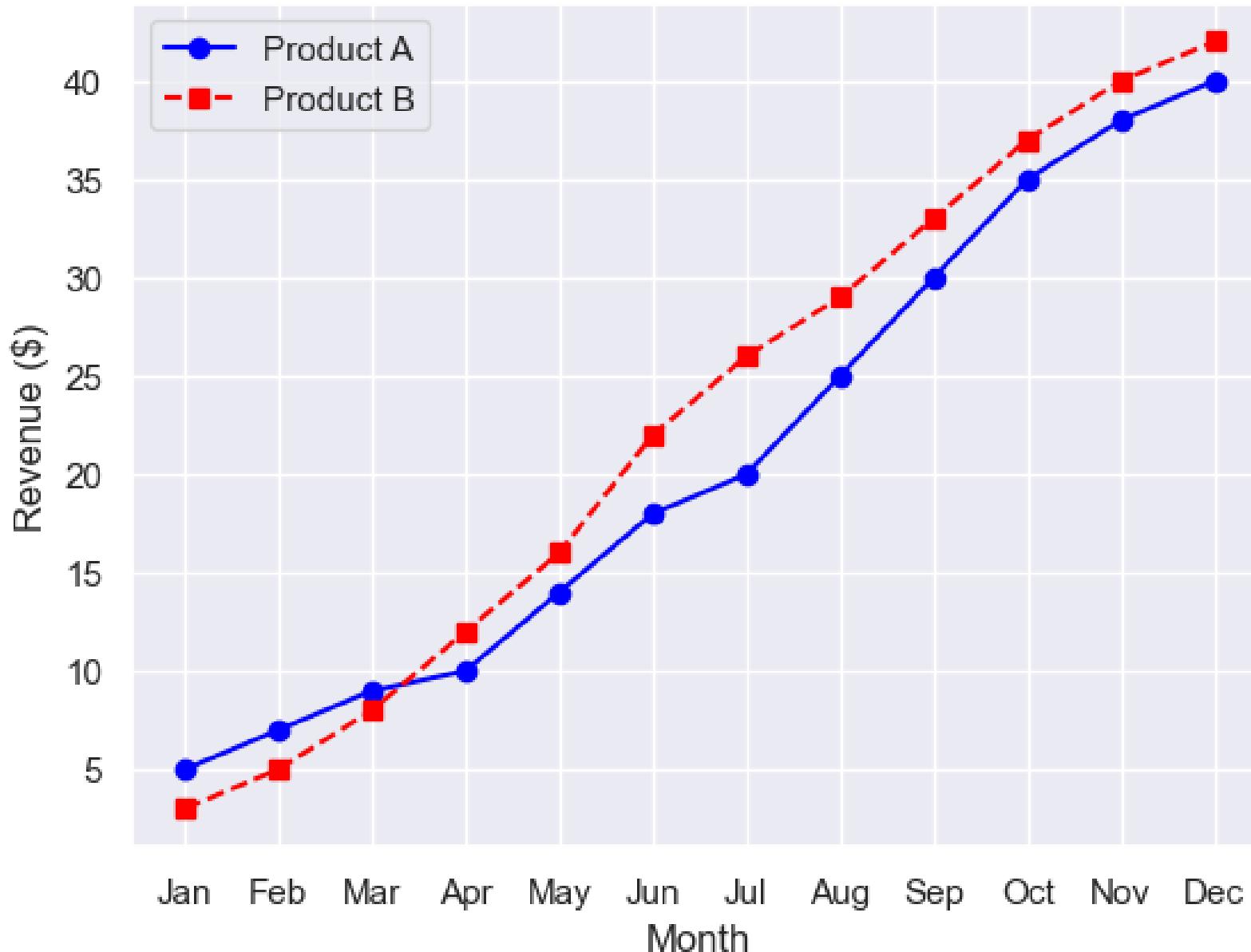
Stock Price Movement



Customer Growth Trend



Revenue Comparison: Product A vs. Product B



Basic Plots in Matplotlib – line plot

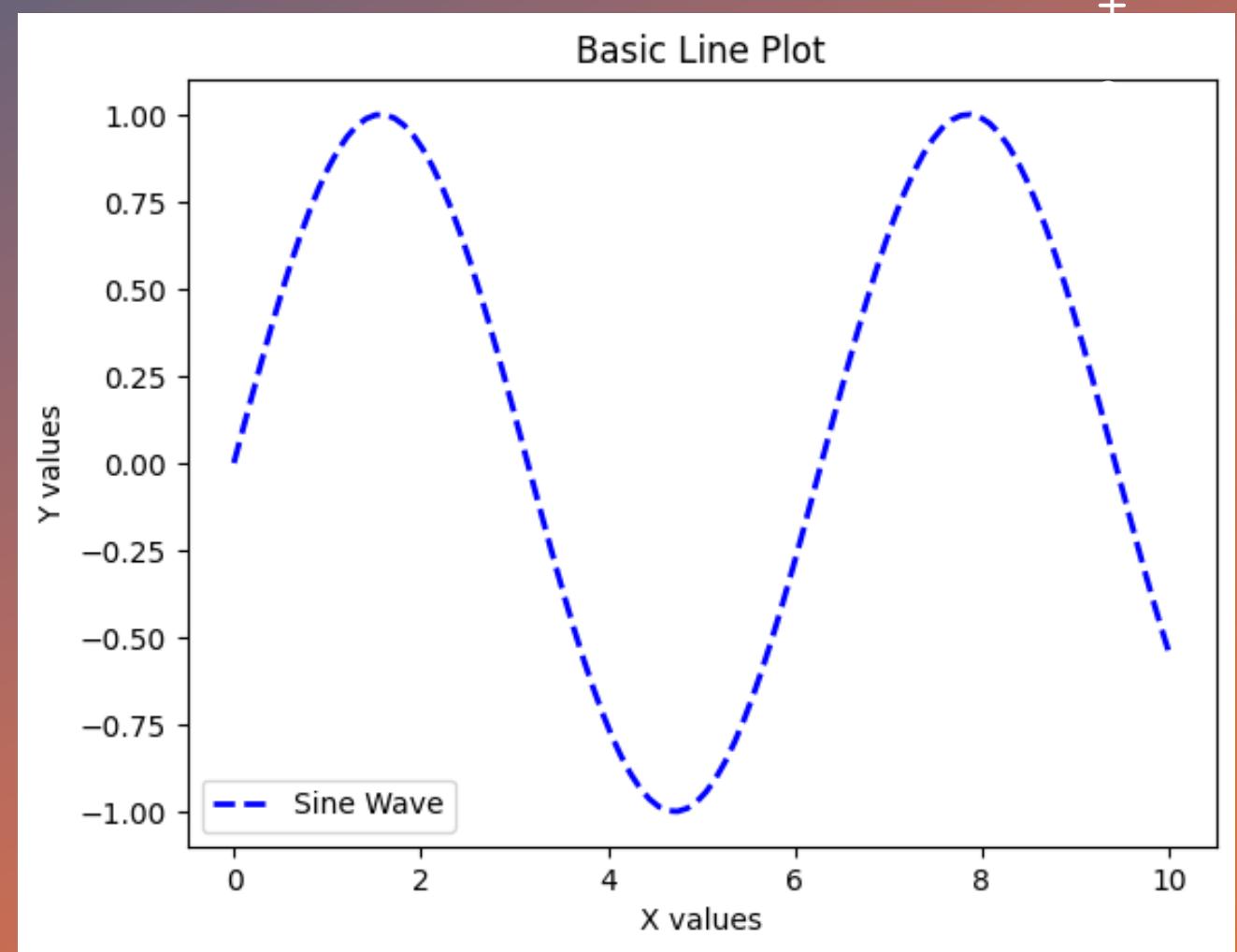
```
# Sample data
x = np.linspace(0, 10, 100) # Generates 100 values between
0 and 10
y = np.sin(x) # Compute sine of x

# Create plot
plt.plot(x, y, label="Sine Wave", color="blue", linestyle="--",
linewidth=2)

# Add labels and title
plt.xlabel("X values")
plt.ylabel("Y values")
plt.title("Basic Line Plot")
plt.legend() # Display legend

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

Basic Plots in Matplotlib – line plot



Basic Plots in Matplotlib – line plot

- Key Functions Used:
- `plt.plot()` → Creates the line
- `plot=plt.xlabel()` & `plt.ylabel()` → Labels the axes.
- `plt.title()` → Adds a title.
- `plt.legend()` → Shows the legend.

plt.plot()

- plt.plot(x, y, marker="o", markersize=8, markerfacecolor="red", markeredgecolor="black", markeredgewidth=2)
- markersize=8: Increases the size of the markers.
- markerfacecolor="red": Fills markers with red color.
- markeredgecolor="black": Sets marker border color to black.
- markeredgewidth=2: Adjusts the thickness of the marker border.
- plt.plot(x, y, marker="s") # Square markers
- plt.plot(x, y, marker="d") # Diamond markers
- plt.plot(x, y, marker="x") # X-shaped markers
- plt.plot(x, y, marker="*") # Star markers

plt.xlabel() & plt.ylabel()

- plt.xlabel("X Axis", fontsize=14, color="blue", labelpad=10)
- plt.ylabel("Y Axis", fontsize=14, color="red", labelpad=15)
 - fontsize=14: Adjusts the font size.
 - color="blue": Changes label color.
 - labelpad=10: Adds space between the label and the axis.

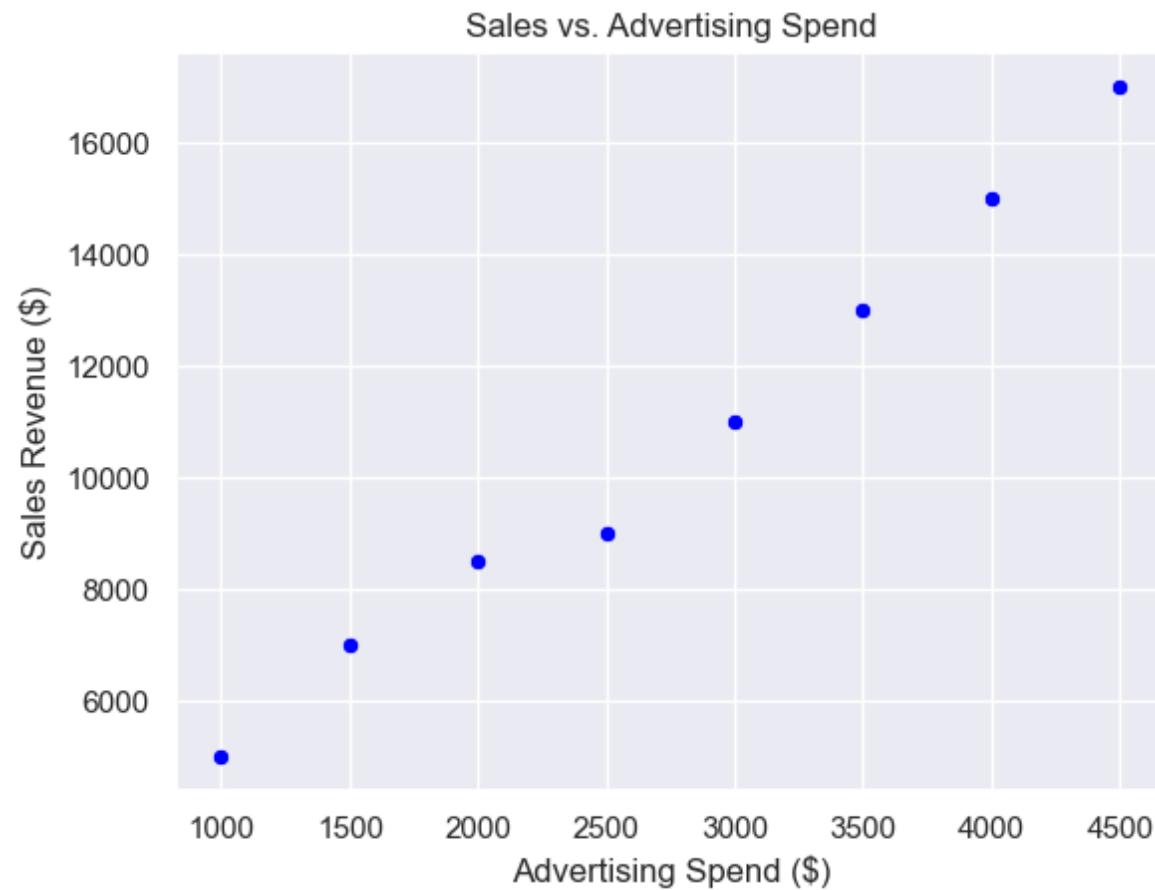
plt.title()

- plt.title("Sine Wave Plot", fontsize=16, color="blue", pad=20, loc="left")
 - fontsize=16: Increases font size.
 - color="blue": Sets title color.
 - pad=20: Adds space between the title and the plot.
 - loc="left": Positions title on the left ("center" by default, "right" is also an option).

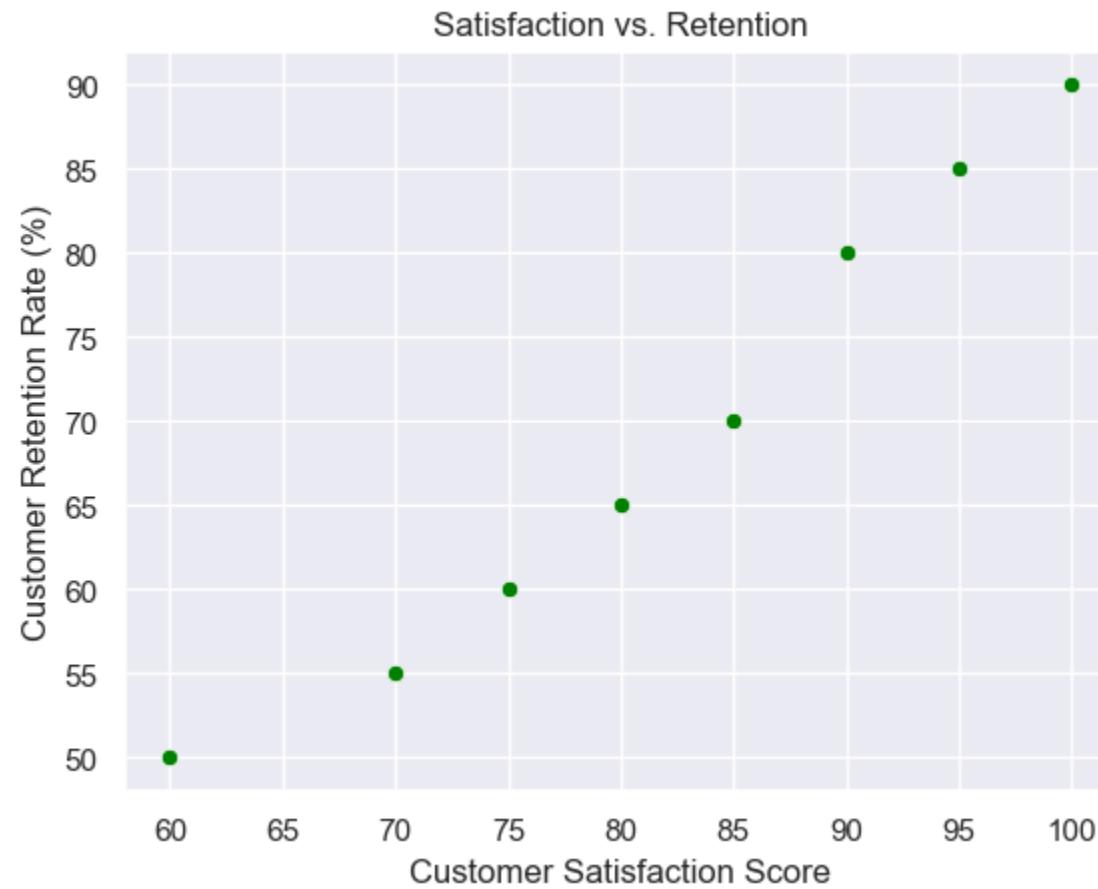
plt.legend()

- plt.legend(loc="upper right", fontsize=12, title="Functions", frameon=True)
 - loc="upper right": Places the legend in the upper-right corner (other options: "upper left", "lower right", etc.).
 - fontsize=12: Adjusts the font size.
 - title="Functions": Adds a title inside the legend.
 - frameon=True: Displays a border around the legend (use False to remove it).

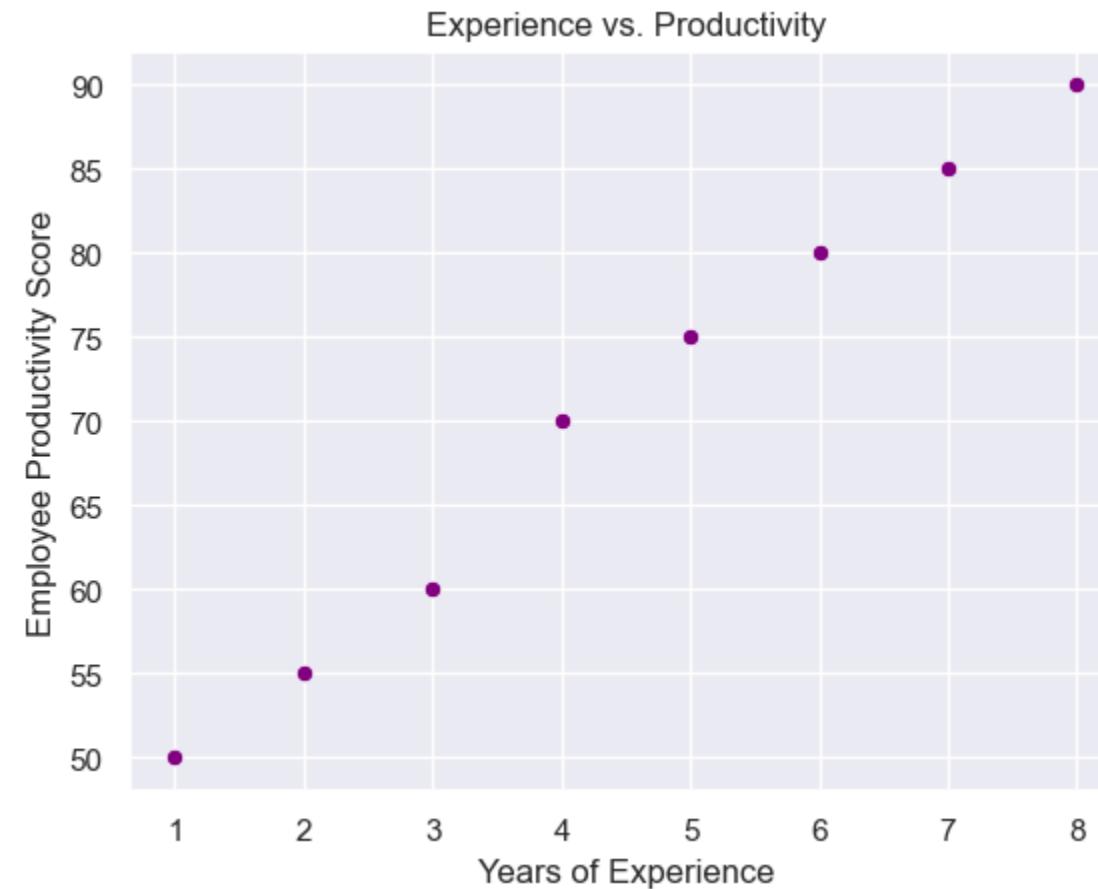
Scatter plot



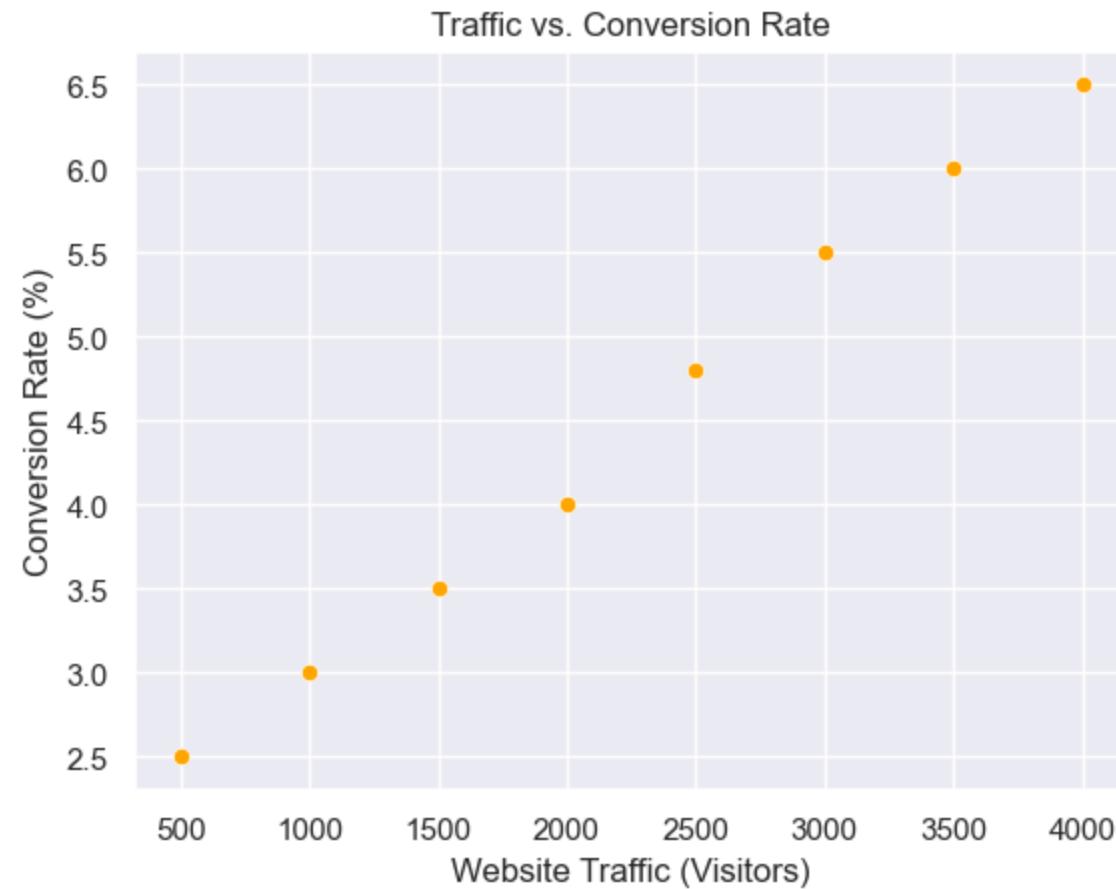
Customer satisfaction vs customer retention



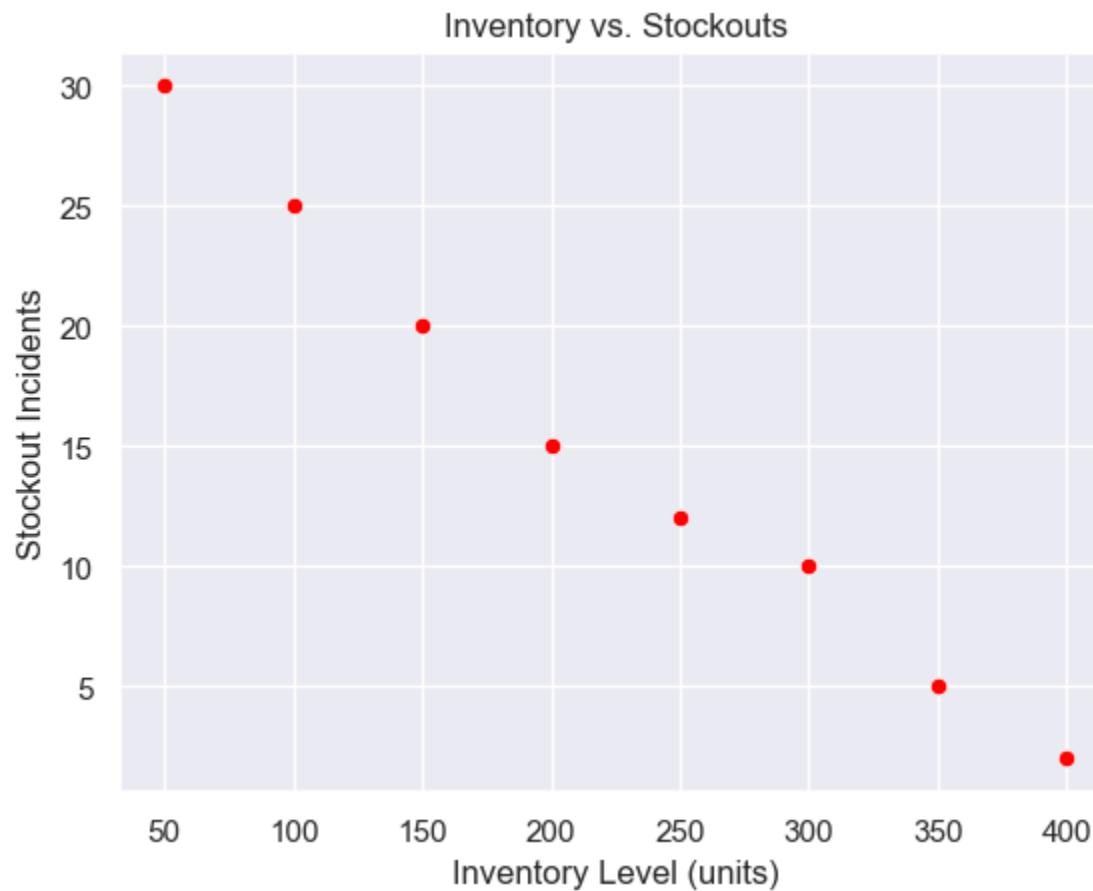
Employee experience vs productivity



Website traffic vs conversion rate



Inventory vs stockouts



Scatter Plot - visualizing relationships between two numerical variables

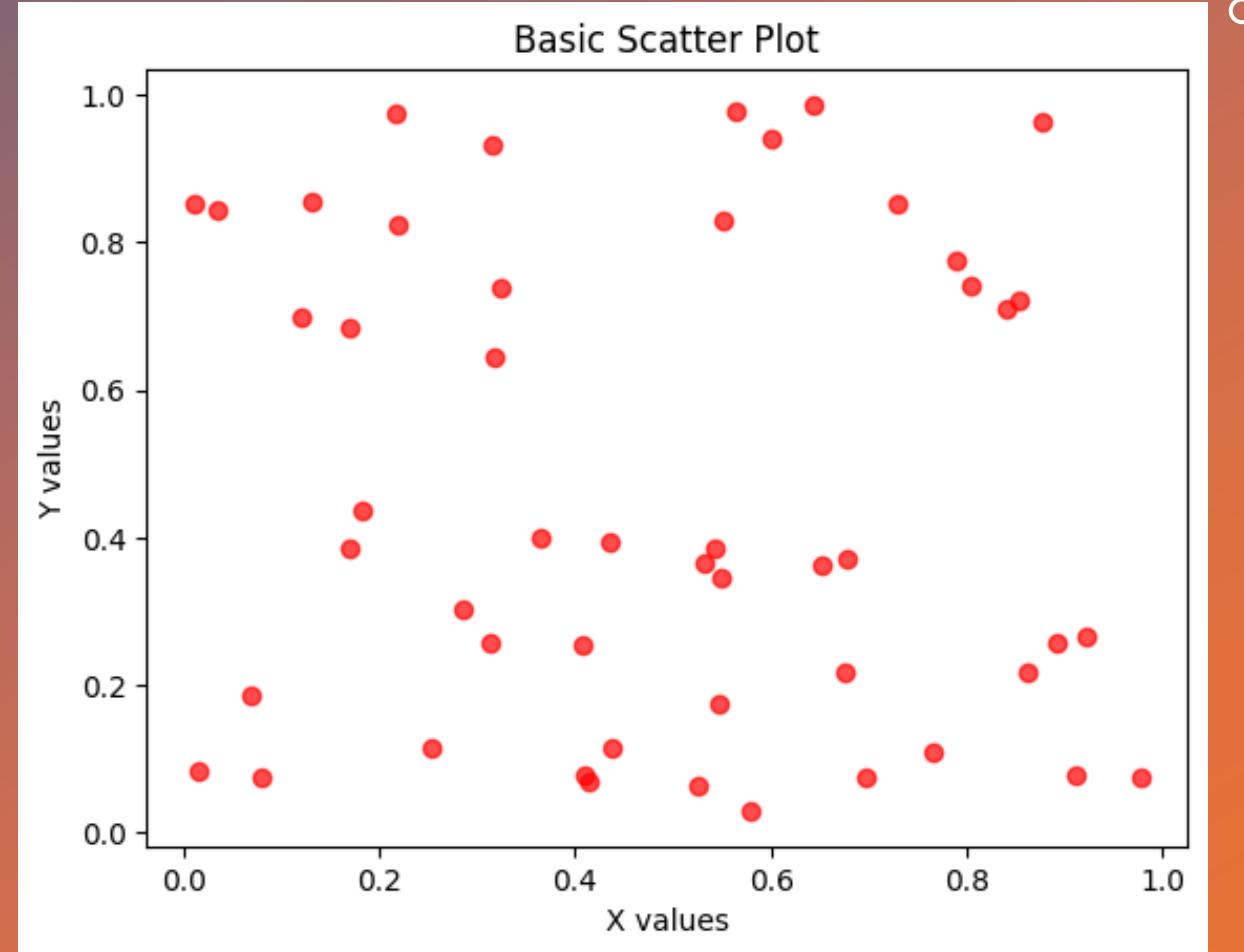
```
# Generate random data
x = np.random.rand(50) # 50 random values
y = np.random.rand(50)

# Create scatter plot
plt.scatter(x, y, color="red", marker="o", alpha=0.7)

# Labels and title
plt.xlabel("X values")
plt.ylabel("Y values")
plt.title("Basic Scatter Plot")

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

Scatter Plot - visualizing relationships between two numerical variables



Customizing scatter plot

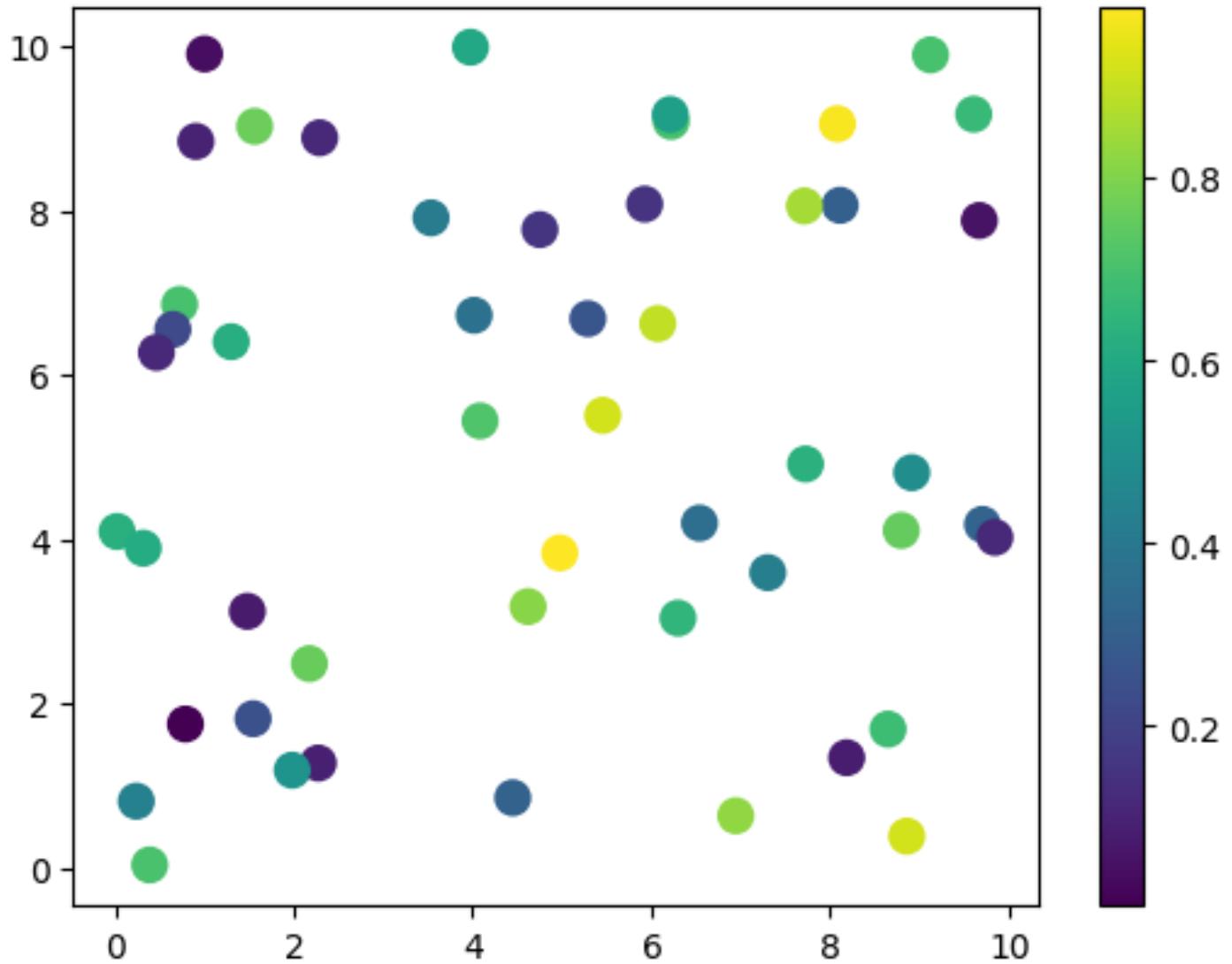
- `plt.scatter(x, y, s=100, c="red", alpha=0.5, marker="o", edgecolors="black", linewidth=1.5)`
 - `s=100`: Adjusts the marker size.
 - `c="red"`: Sets the color to red.
 - `alpha=0.5`: Adds transparency (0 = fully transparent, 1 = fully opaque).
 - `marker="o"`: Uses circle markers (you can change to "s" for squares, "^" for triangles, etc.).
 - `edgecolors="black"`: Adds black borders to markers.
 - `linewidth=1.5`: Adjusts the border thickness.

Customizing scatter plot

- `colors = np.random.rand(50) # Random color values`
- `plt.scatter(x, y, c=colors, cmap="viridis", s=100)`
 - `plt.title("Scatter Plot with Colormap")`
 - `plt.show()`
 - `c=colors`: Assigns different colors to each point.
 - `cmap="viridis"`: Uses the Viridis colormap (other options: "plasma", "coolwarm", "rainbow").
 - `plt.colorbar()`: Adds a color scale legend.

Output

Scatter Plot with Colormap



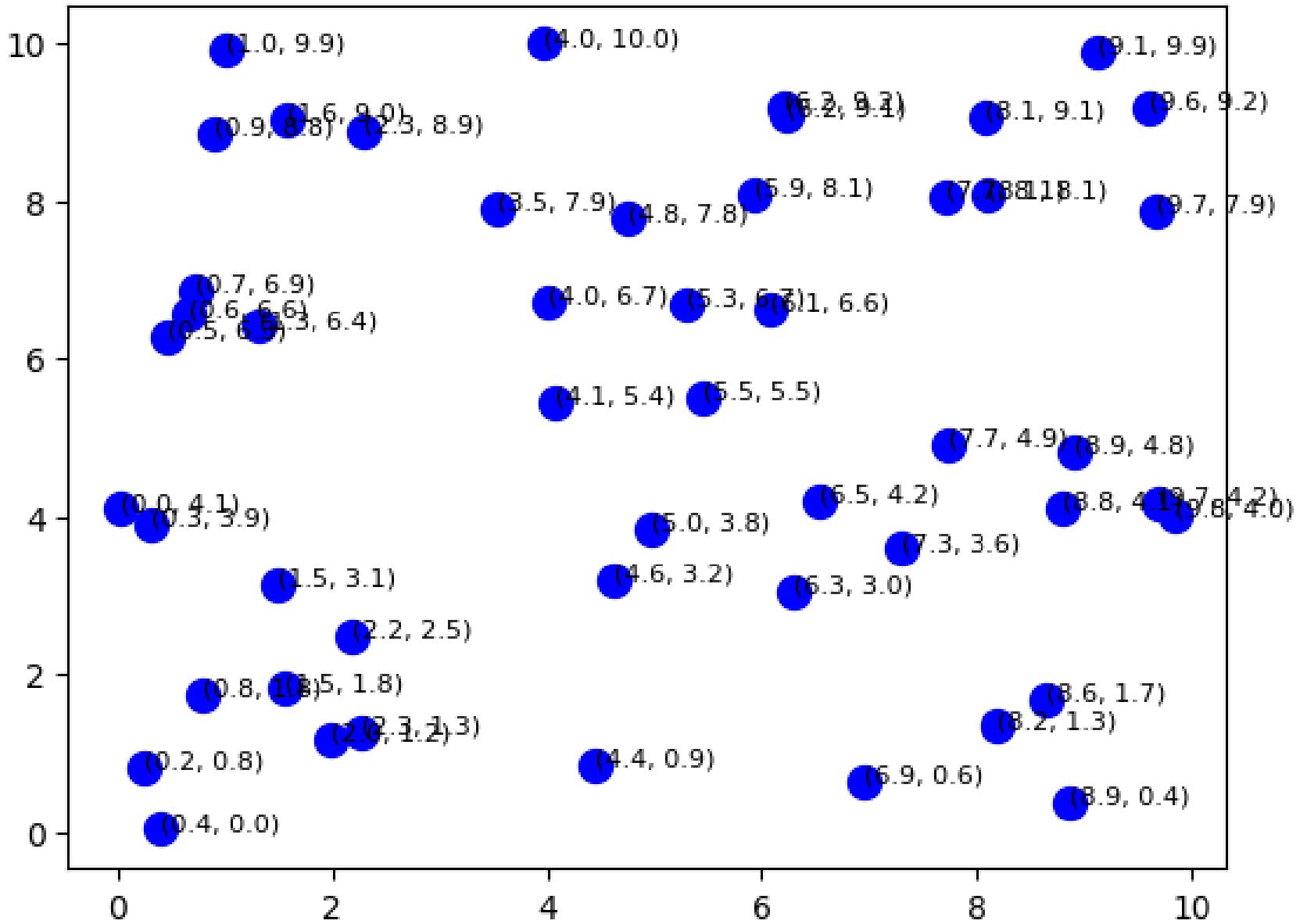
Adding Labels to Points

- for i in range(len(x)):
- plt.text(x[i], y[i], f"({x[i]:.1f}, {y[i]:.1f})", fontsize=8)

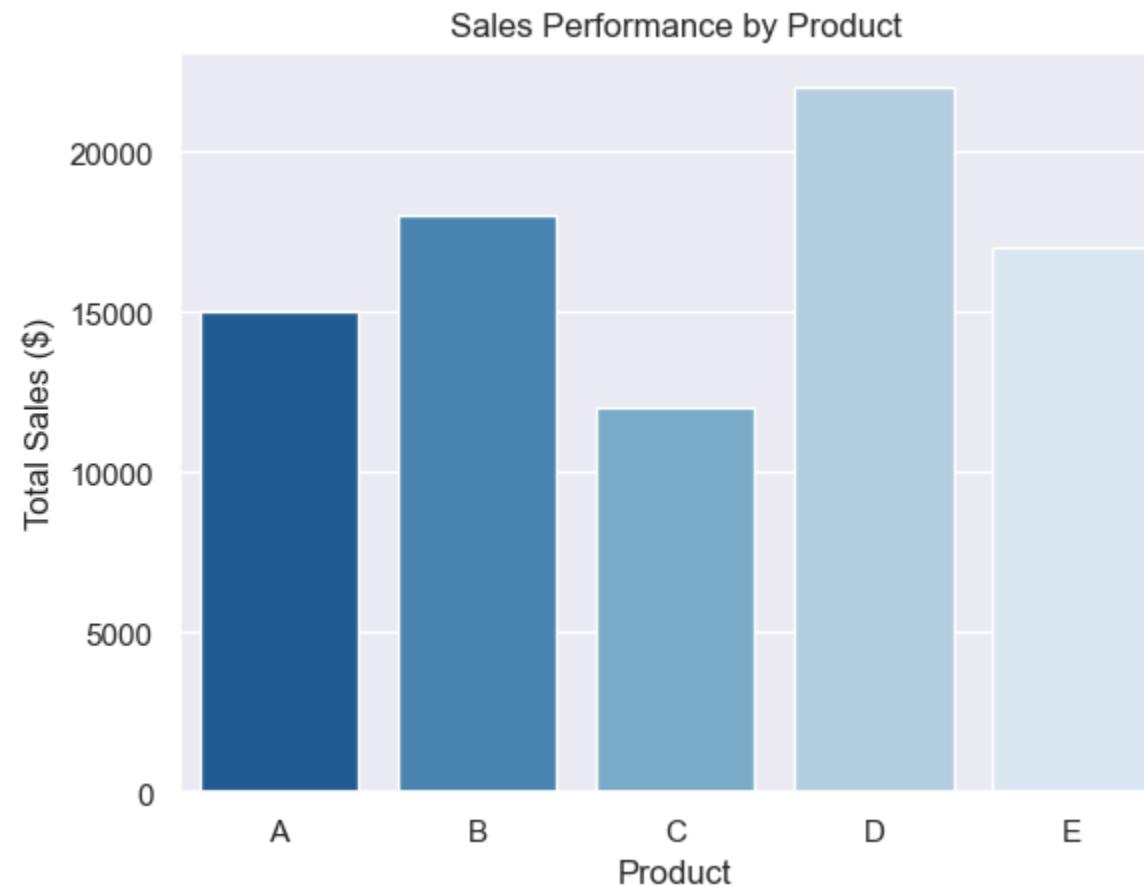
- plt.scatter(x, y, s=100, c="blue")
- plt.title("Scatter Plot with Labels")
- plt.show()
 - plt.text(x[i], y[i], "label"): Places text near each point.

Output

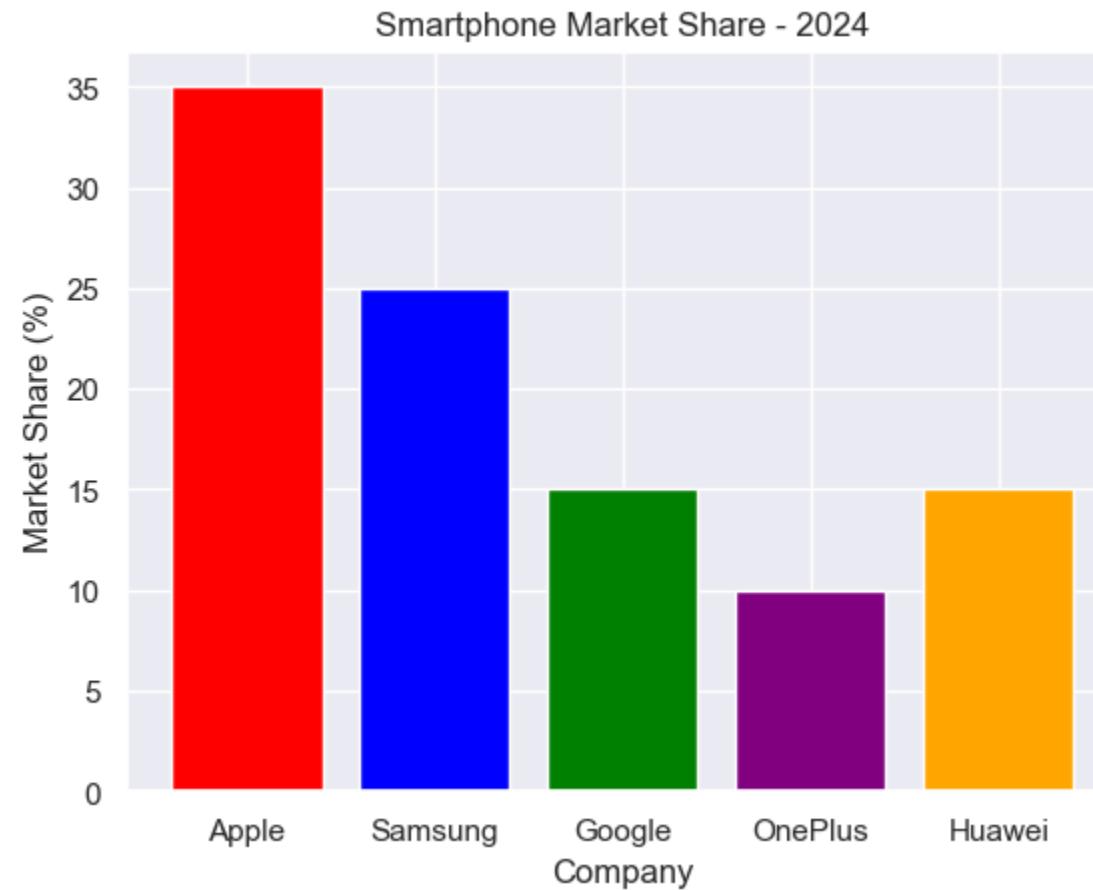
Scatter Plot with Labels



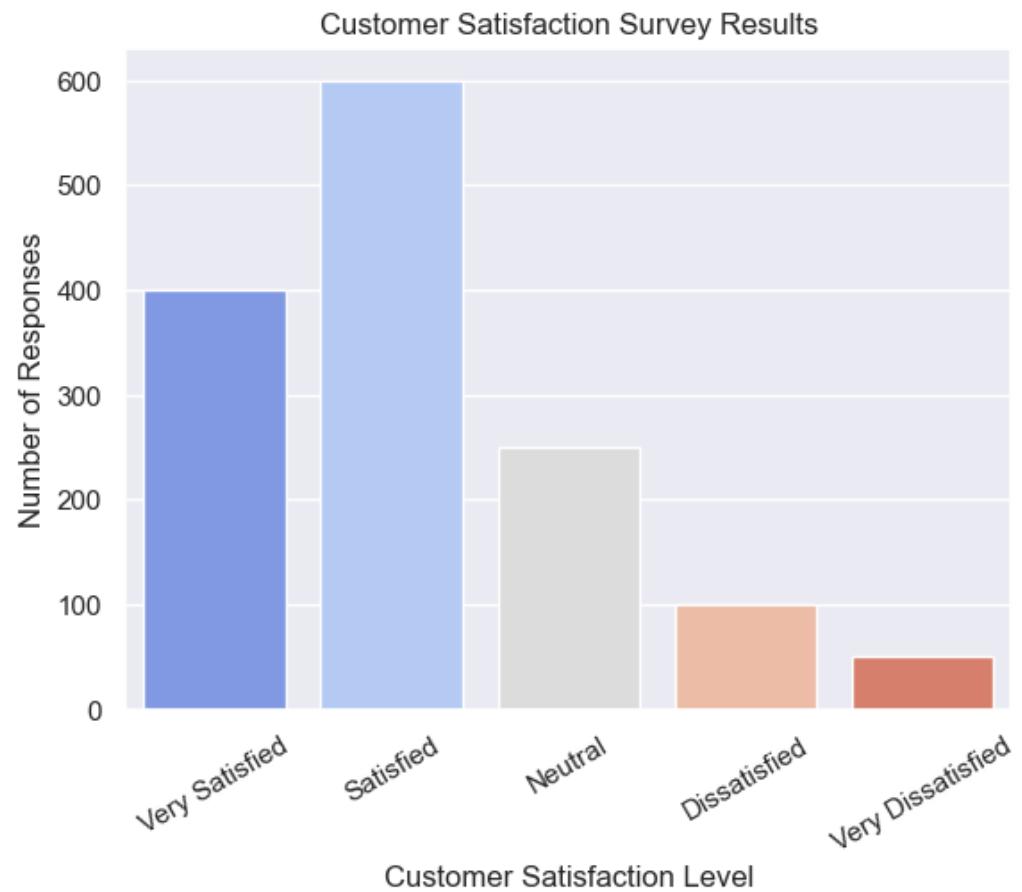
Bar charts – sales performance by product



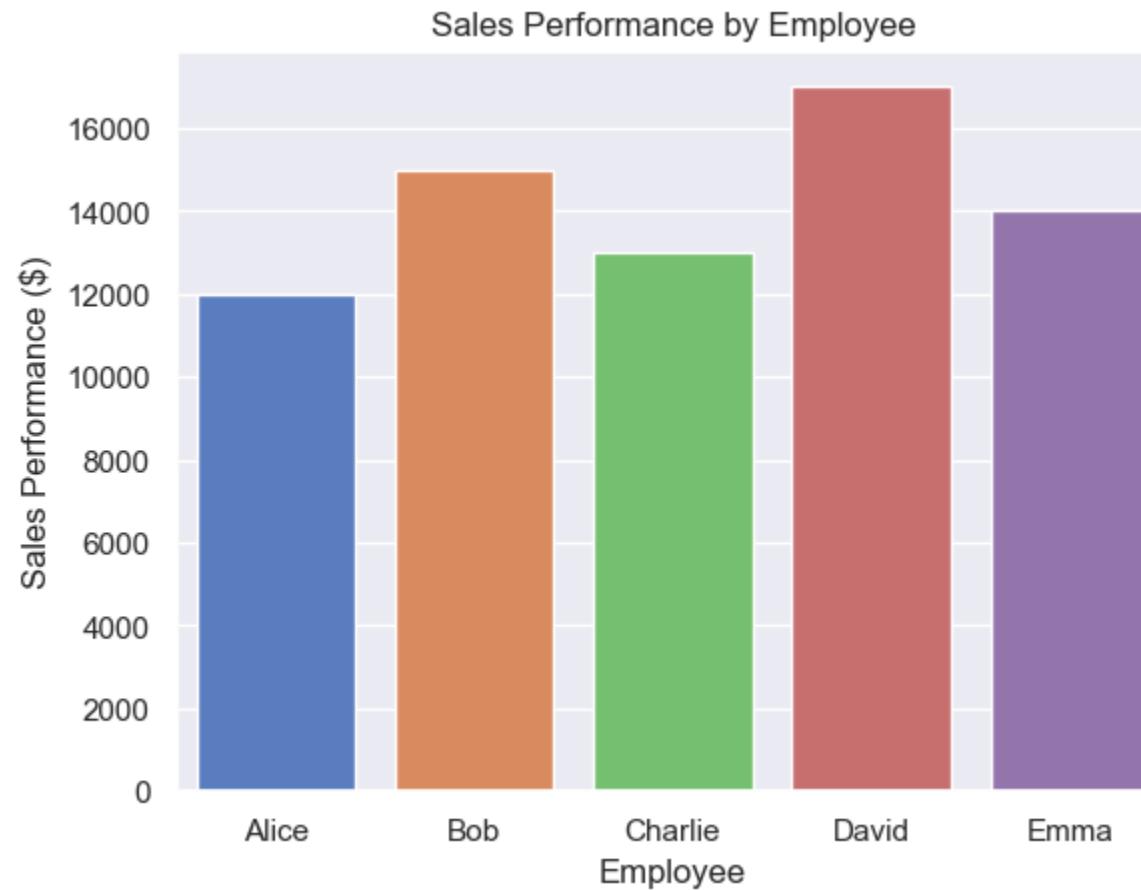
Smart phone market share by company



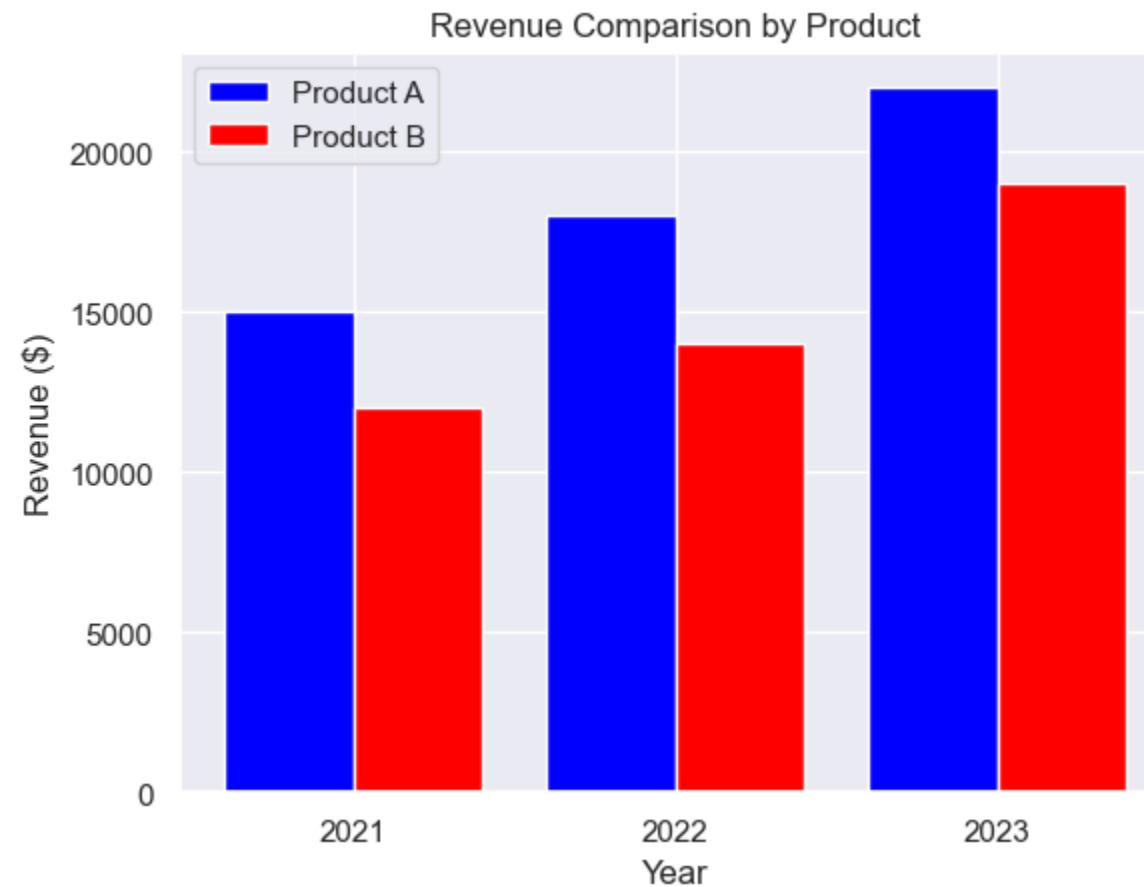
Customer survey



Employee performance comparison



Advanced – product comparison by year



Bar Chart - to compare categorical data

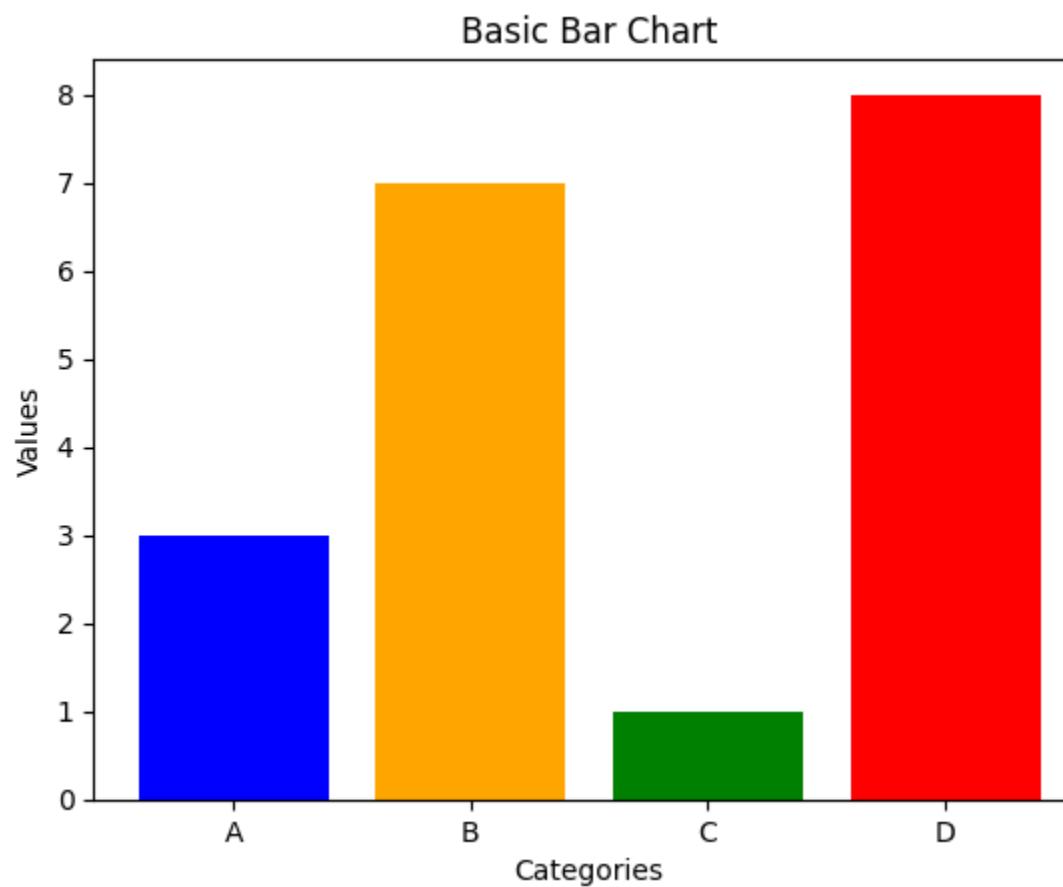
```
# Data
categories = ["A", "B", "C", "D"]
values = [3, 7, 1, 8]

# Create bar chart
plt.bar(categories, values, color=["blue", "orange", "green",
"red"])

# Labels and title
plt.xlabel("Categories")
plt.ylabel("Values")
plt.title("Basic Bar Chart")

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

Bar Chart

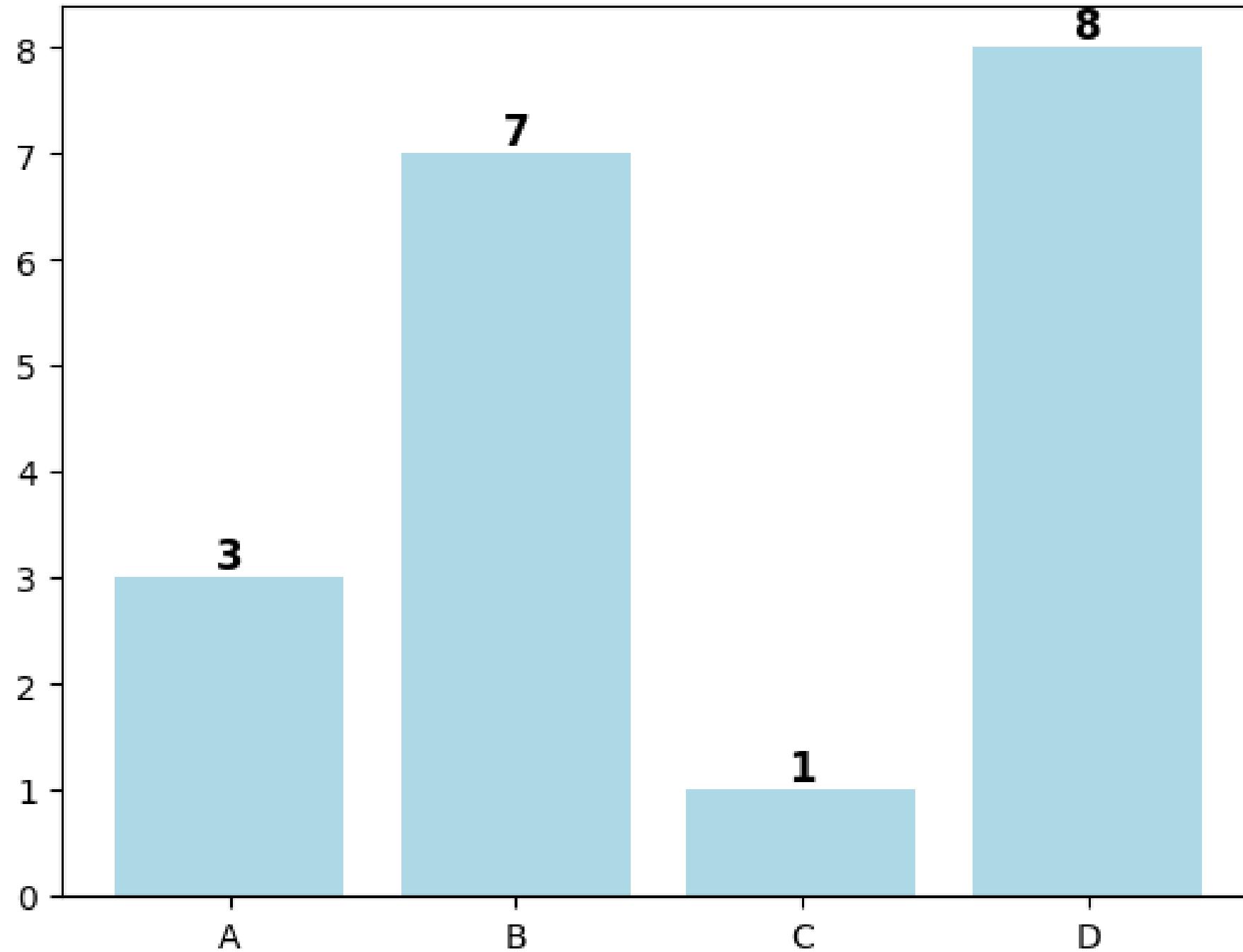


Customizing Bar charts

- Adding Edge Color & Line Width
 - `plt.bar(categories, values, color="skyblue", edgecolor="black", linewidth=2)`
 - `edgecolor="black"`: Adds a black border around bars.
 - `linewidth=2`: Increases border thickness.

Displaying Values on Bars

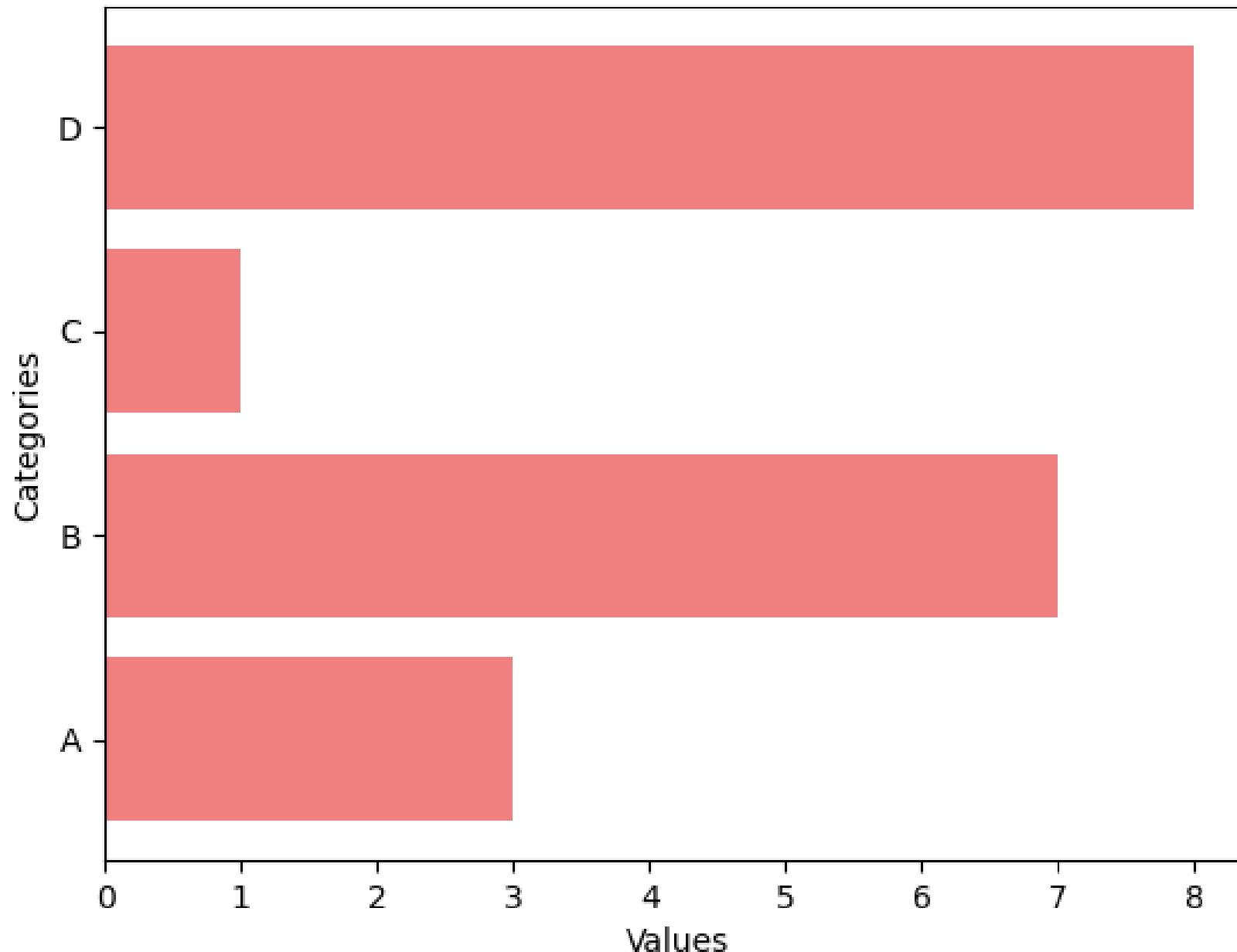
- bars = plt.bar(categories, values, color="lightblue")
Add text on bars
- for bar in bars:
 - plt.text(bar.get_x() + bar.get_width()/2, bar.get_height(), f"{bar.get_height()}")
 - ha="center", va="bottom", fontsize=12, fontweight="bold")
 - bar.get_x() + bar.get_width()/2: Positions text at the center of each bar.
 - bar.get_height(): Places text just above the bar.



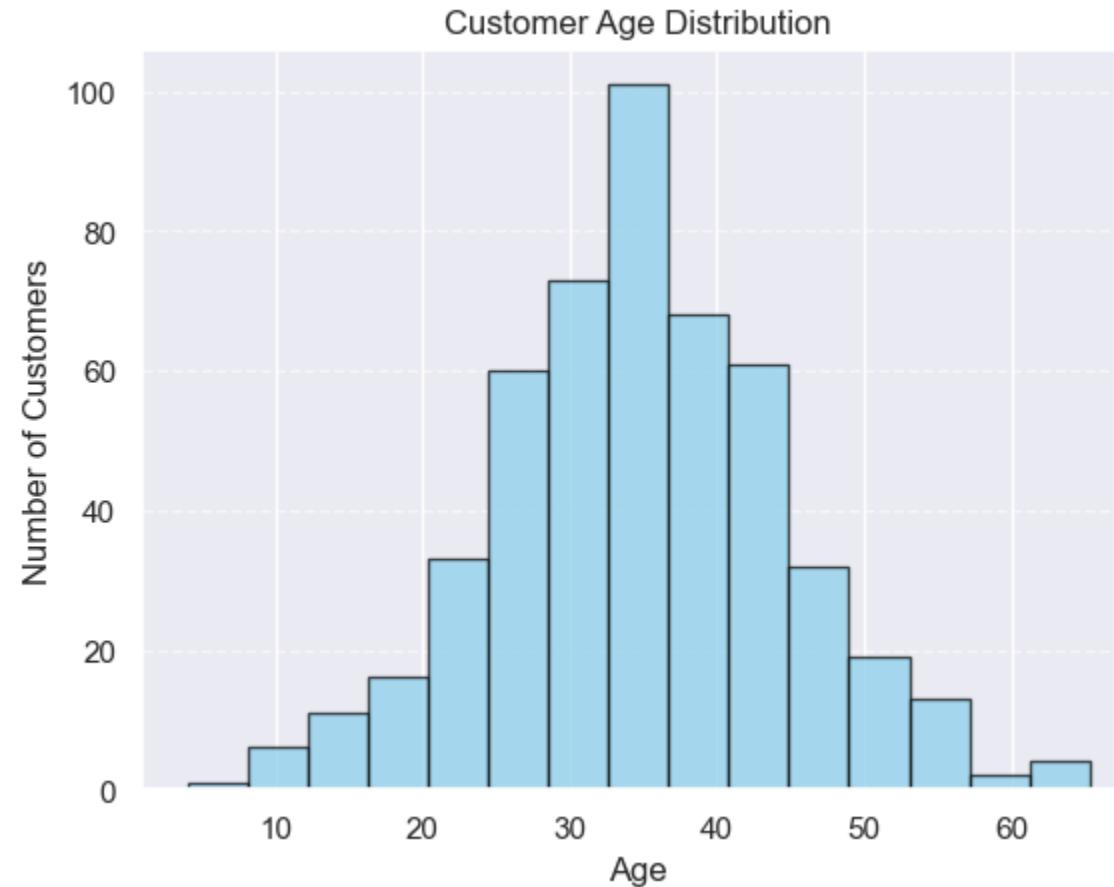
Horizontal Bar Chart

- plt.barh(categories, values, color="lightcoral")
- plt.xlabel("Values")
- plt.ylabel("Categories")
- plt.title("Horizontal Bar Chart")
- plt.show()

Horizontal Bar Chart



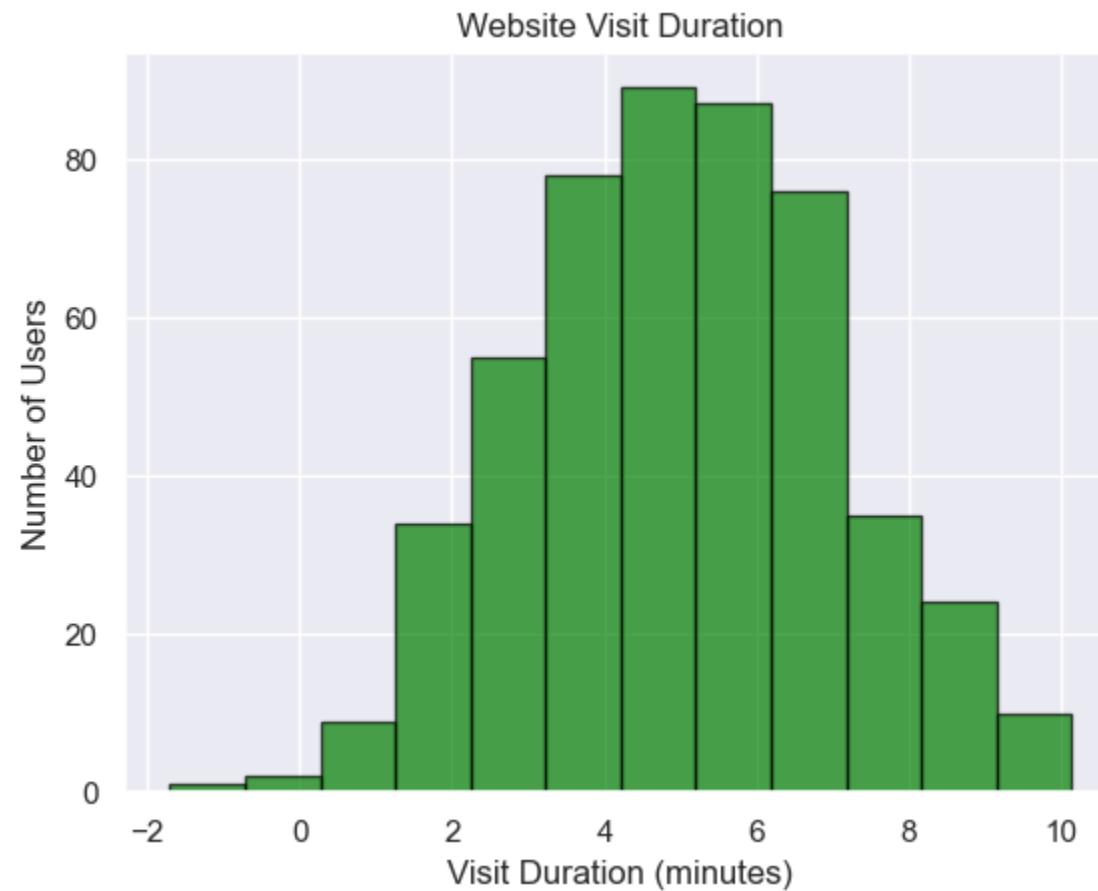
Histogram examples – Customer age distribution



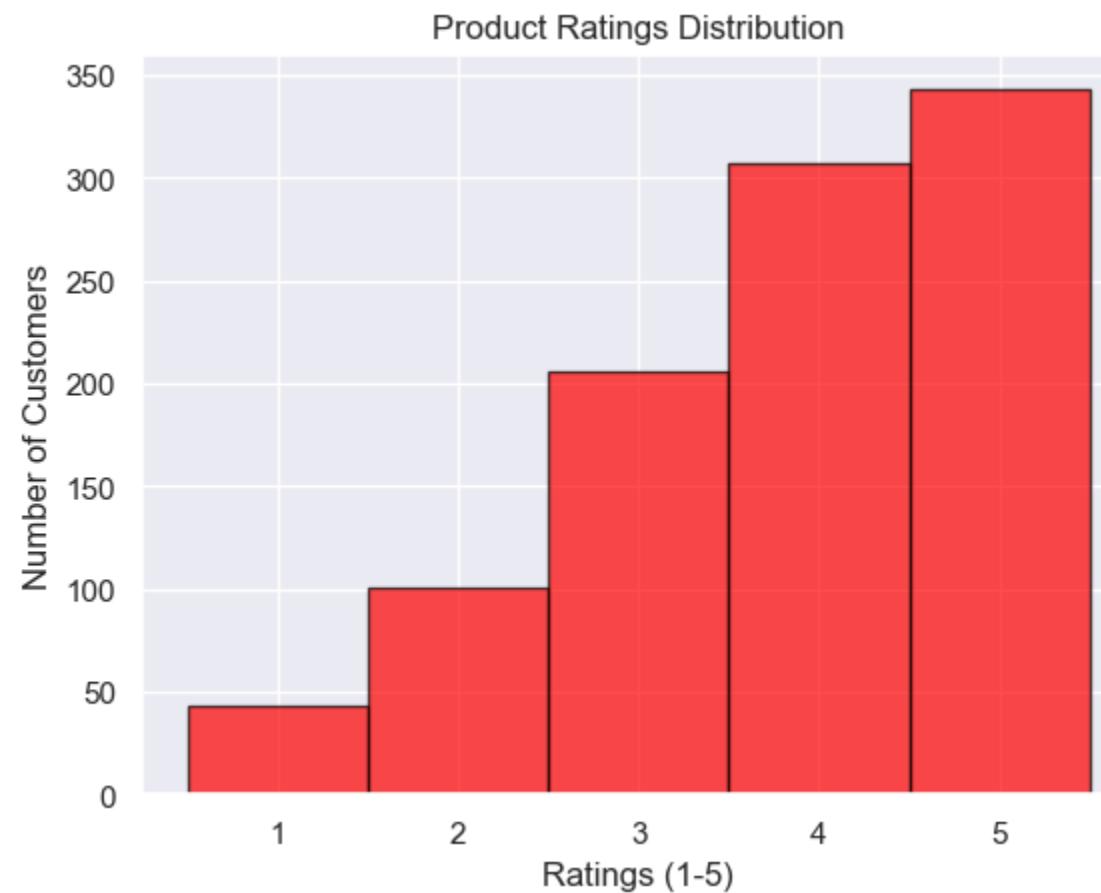
Sales revenue distribution – high end vs low end products



Website visit duration



Product ratings



Histogram - distribution of a dataset

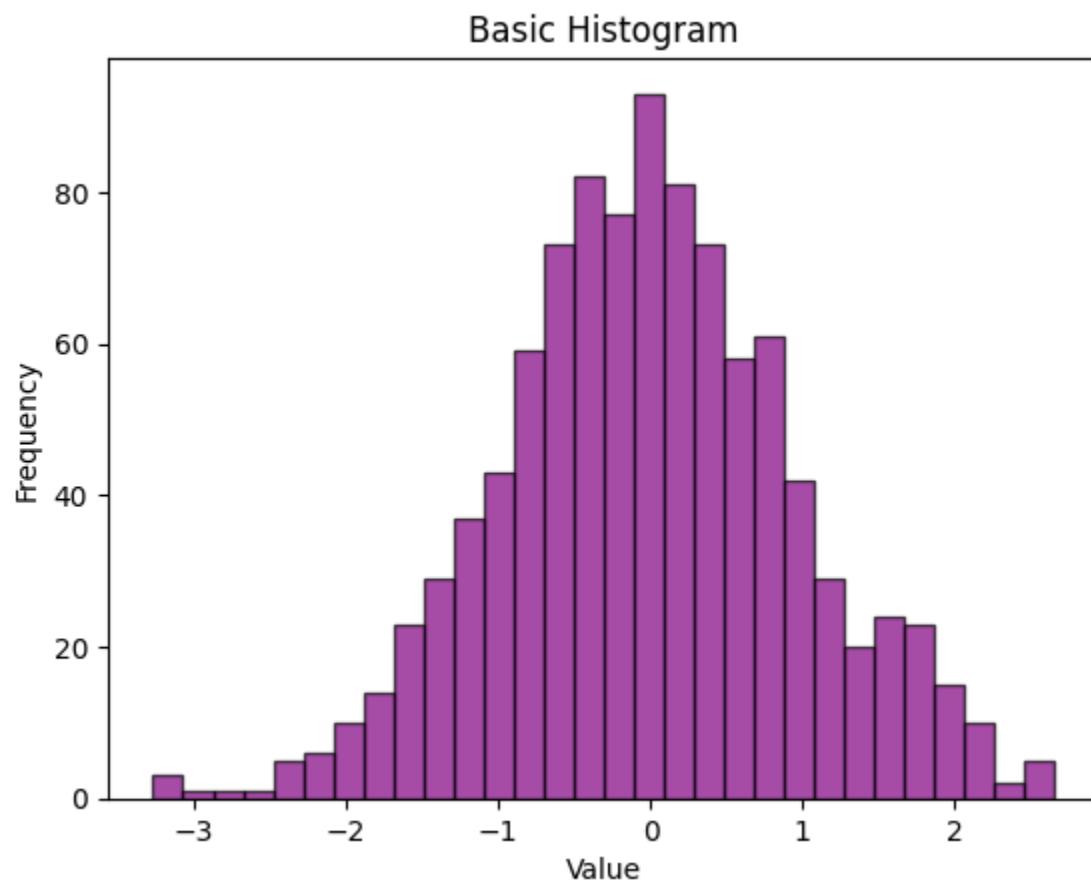
```
# Generate random data
data = np.random.randn(1000) # 1000 random numbers from normal distribution

# Create histogram
plt.hist(data, bins=30, color="purple", edgecolor="black", alpha=0.7)

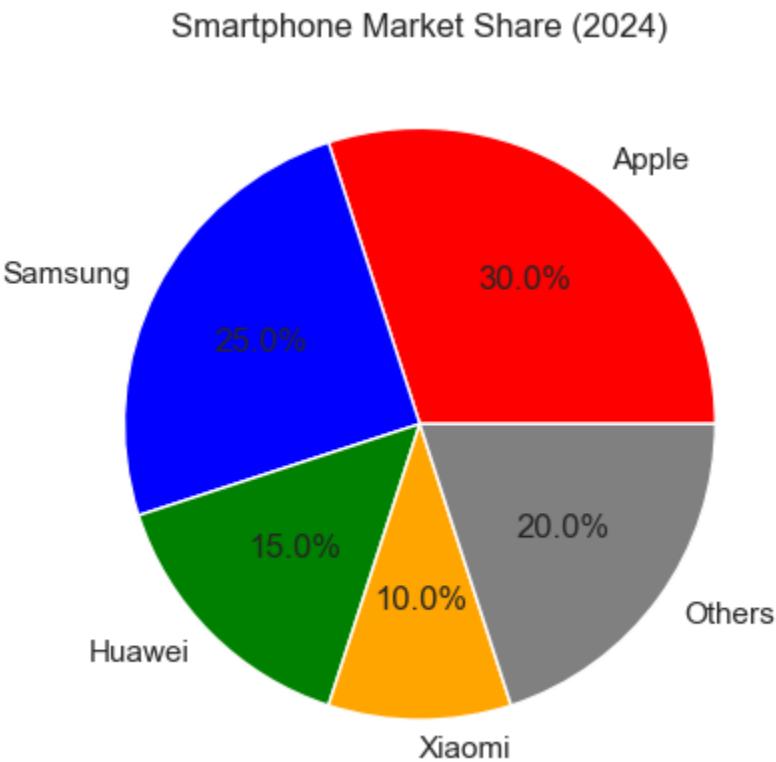
# Labels and title
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.title("Basic Histogram")

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

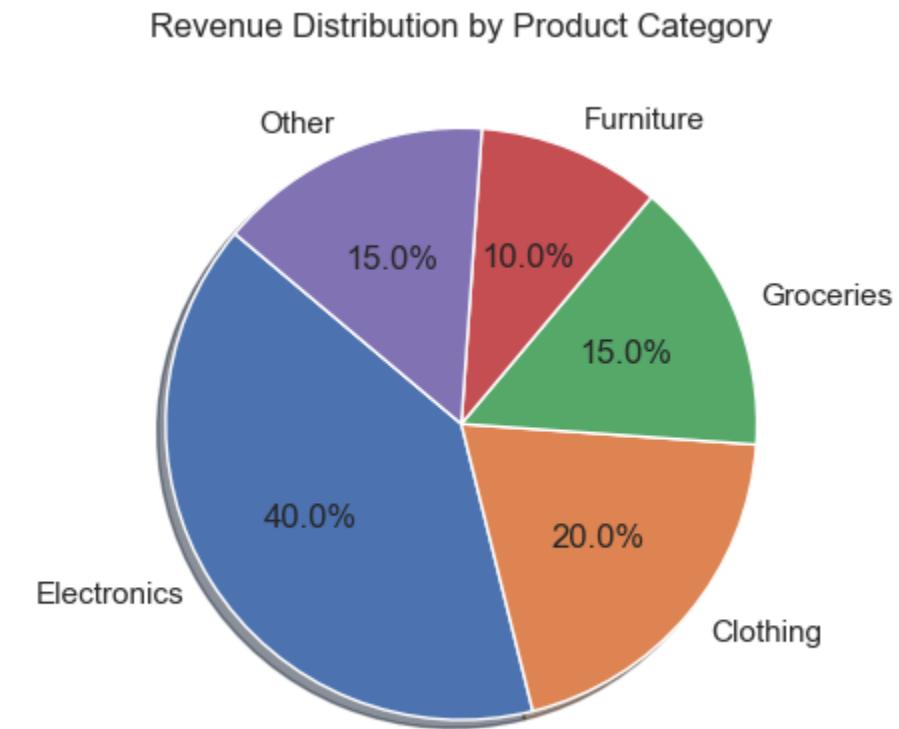
Histogram - distribution of a dataset



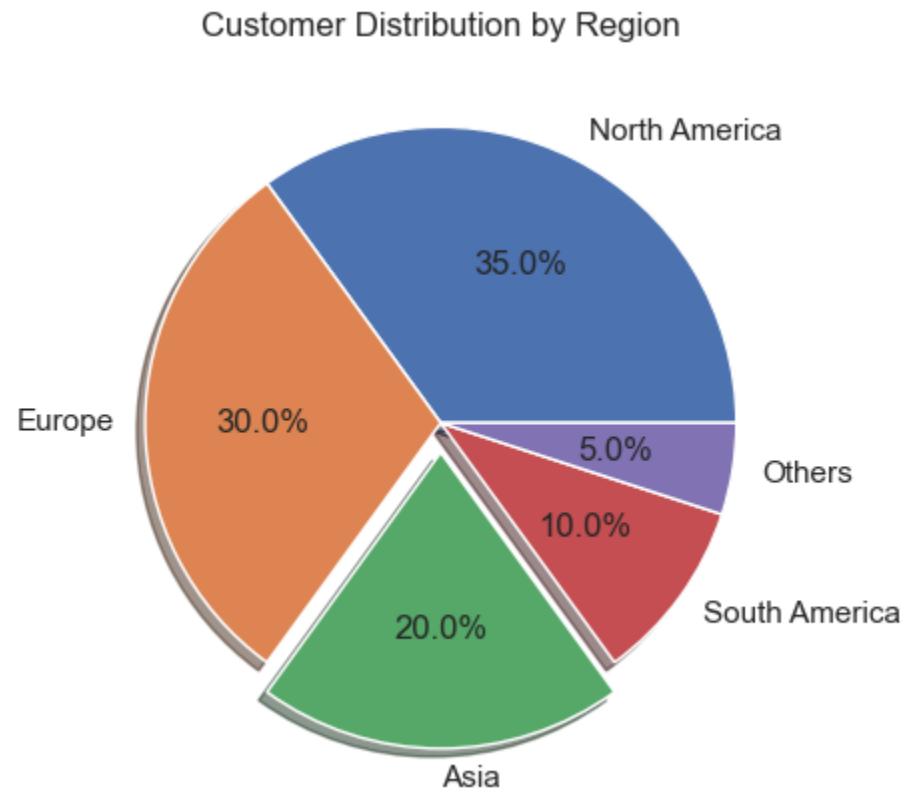
Pie Charts



Revenue by product



Customer distribution by region



Pie Chart - shows proportions

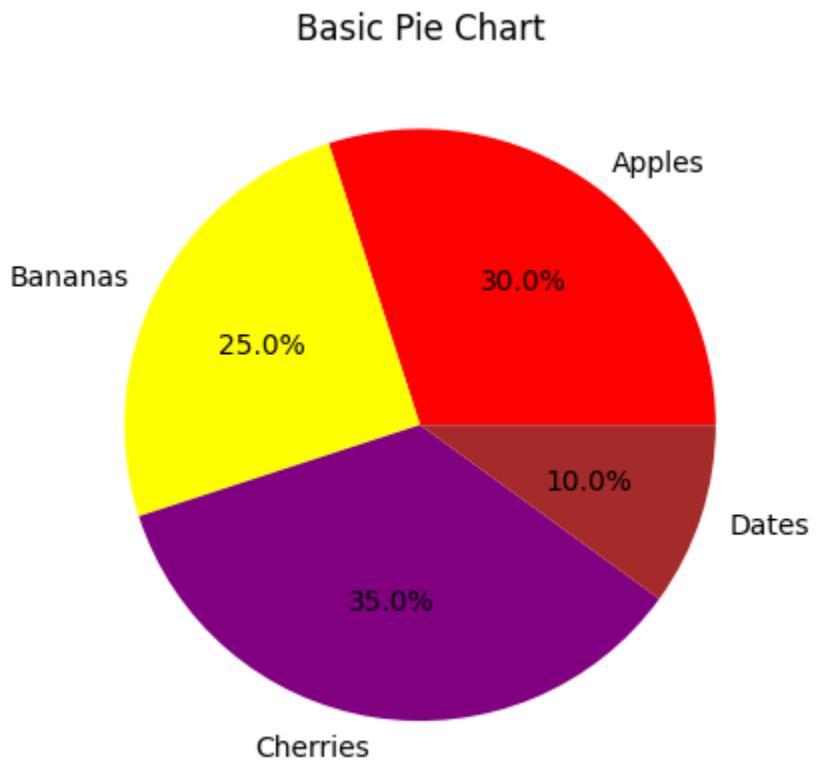
```
# Data
labels = ["Apples", "Bananas", "Cherries", "Dates"]
sizes = [30, 25, 35, 10] # Percentage values

# Create pie chart
plt.pie(sizes, labels=labels, autopct="%1.1f%%", colors=["red", "yellow", "purple", "brown"])

# Title
plt.title("Basic Pie Chart")

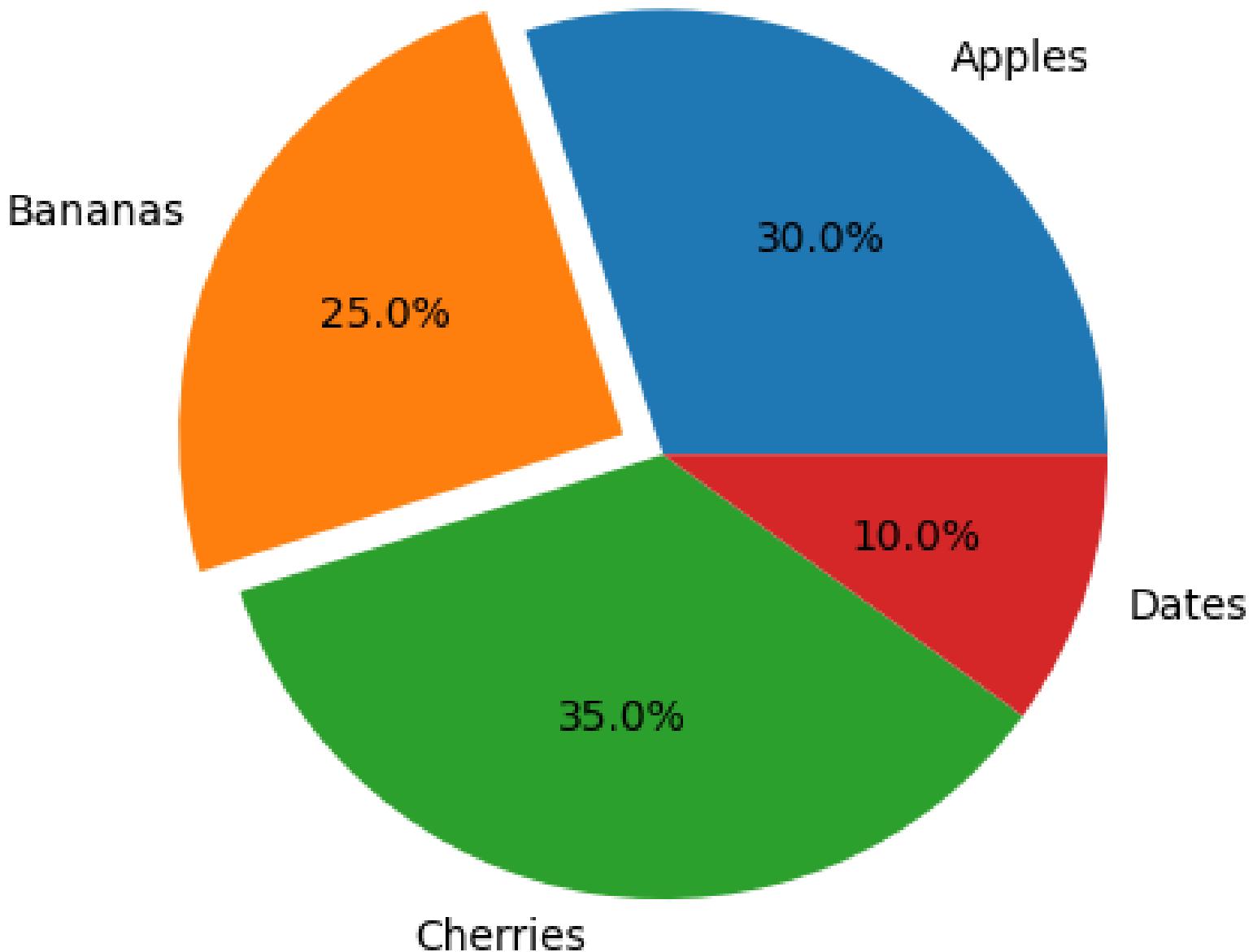
# Show the plot
plt.show()
```

Pie Chart



Exploding a slice

- `explode = (0, 0.1, 0, 0) # Only "Bananas" slice is pulled out`
- `plt.pie(sizes, labels=labels, explode=explode,
autopct="%1.1f%%")`



Combining Multiple Plots

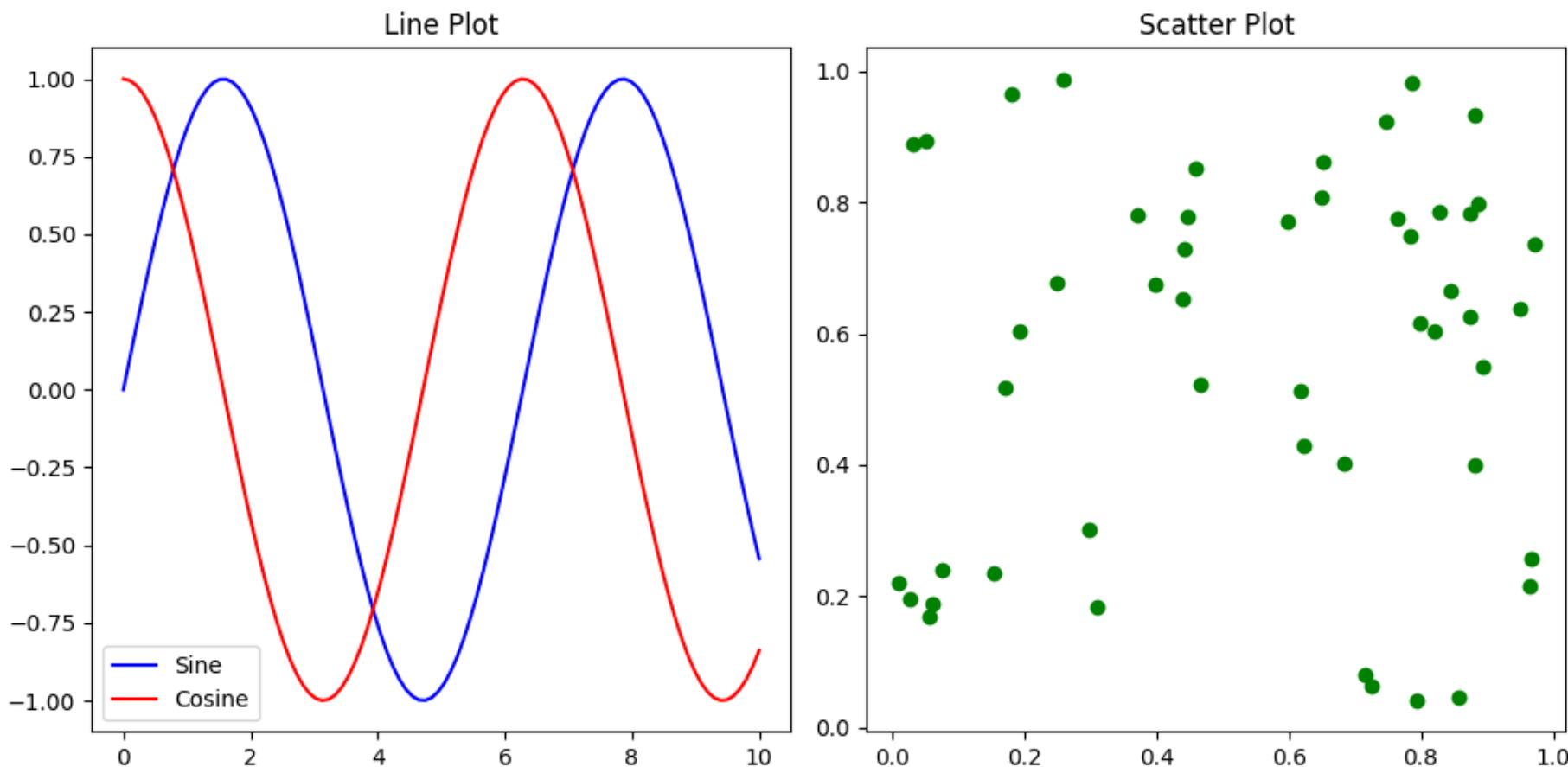
```
# Create a figure with 2 subplots
plt.figure(figsize=(10, 5))

# First subplot (Line Plot)
plt.subplot(1, 2, 1)
x = np.linspace(0, 10, 100)
plt.plot(x, np.sin(x), label="Sine", color="blue")
plt.plot(x, np.cos(x), label="Cosine", color="red")
plt.legend()
plt.title("Line Plot")

# Second subplot (Scatter Plot)
plt.subplot(1, 2, 2)
plt.scatter(np.random.rand(50), np.random.rand(50),
           color="green")
plt.title("Scatter Plot")

# Show the figure
plt.tight_layout()
plt.show()
```

Combining Plots



Saving Plots

- `plt.plot(x, np.sin(x))`
- `plt.title("Saved Plot")`
- `plt.savefig("plot.png", dpi=300)`

Week 6

Visualization with seaborn

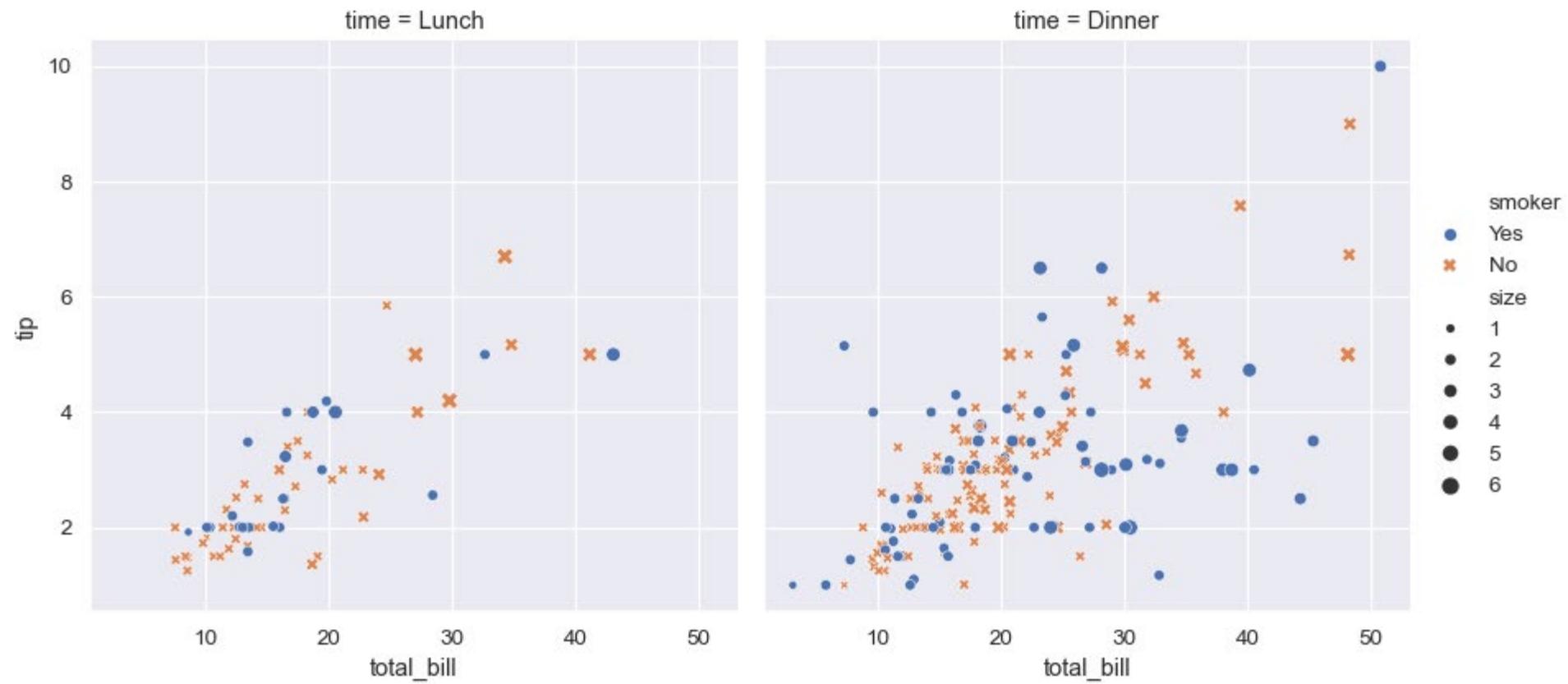
An introduction to seaborn

- Seaborn is a library for making statistical graphics in Python. It builds on top of matplotlib and integrates closely with pandas data structures.
- Seaborn helps you explore and understand your data. Its plotting functions operate on dataframes and arrays containing whole datasets and internally perform the necessary semantic mapping and statistical aggregation to produce informative plots. Its dataset-oriented, declarative API lets you focus on what the different elements of your plots mean, rather than on the details of how to draw them.
- Here's an example of what seaborn can do:

An introduction to seaborn

- Here's an example of what seaborn can do:

```
# Import seaborn  
import seaborn as sns  
  
# Apply the default theme  
sns.set_theme()  
  
# Load an example dataset  
tips = sns.load_dataset("tips")  
  
# Create a visualization  
sns.relplot(data=tips, x="total_bill", y="tip", col="time", hue="smoker",  
style="smoker", size="size")
```



`sns.set_theme()`

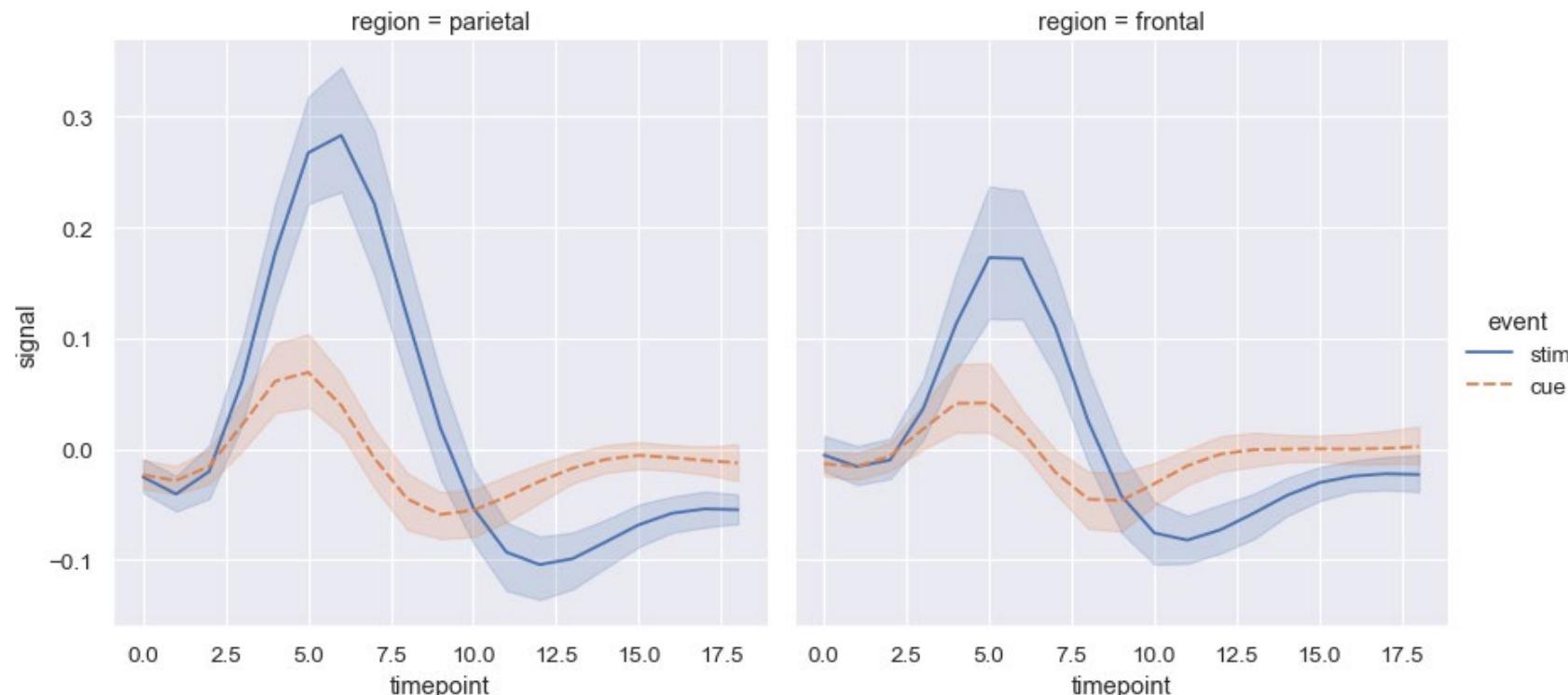
- This will affect how all matplotlib plots look, even if you don't make them with seaborn.
- Beyond the default theme, there are several other options, and you can independently control the style and scaling of the plot to quickly translate your work between presentation.
- `sns.set_theme(style="darkgrid")`
- "darkgrid" → Grid background (good for line plots).
- "whitegrid" → Light grid (useful for statistical plots).
- "dark" → Dark background, no grid.
- "white" → Clean background.
- "ticks" → Minimal style with axis ticks.

```
sns.relplot( data=tips, x="total_bill", y="tip", col="time", hue="smoker", style="smoker", size="size", )
```

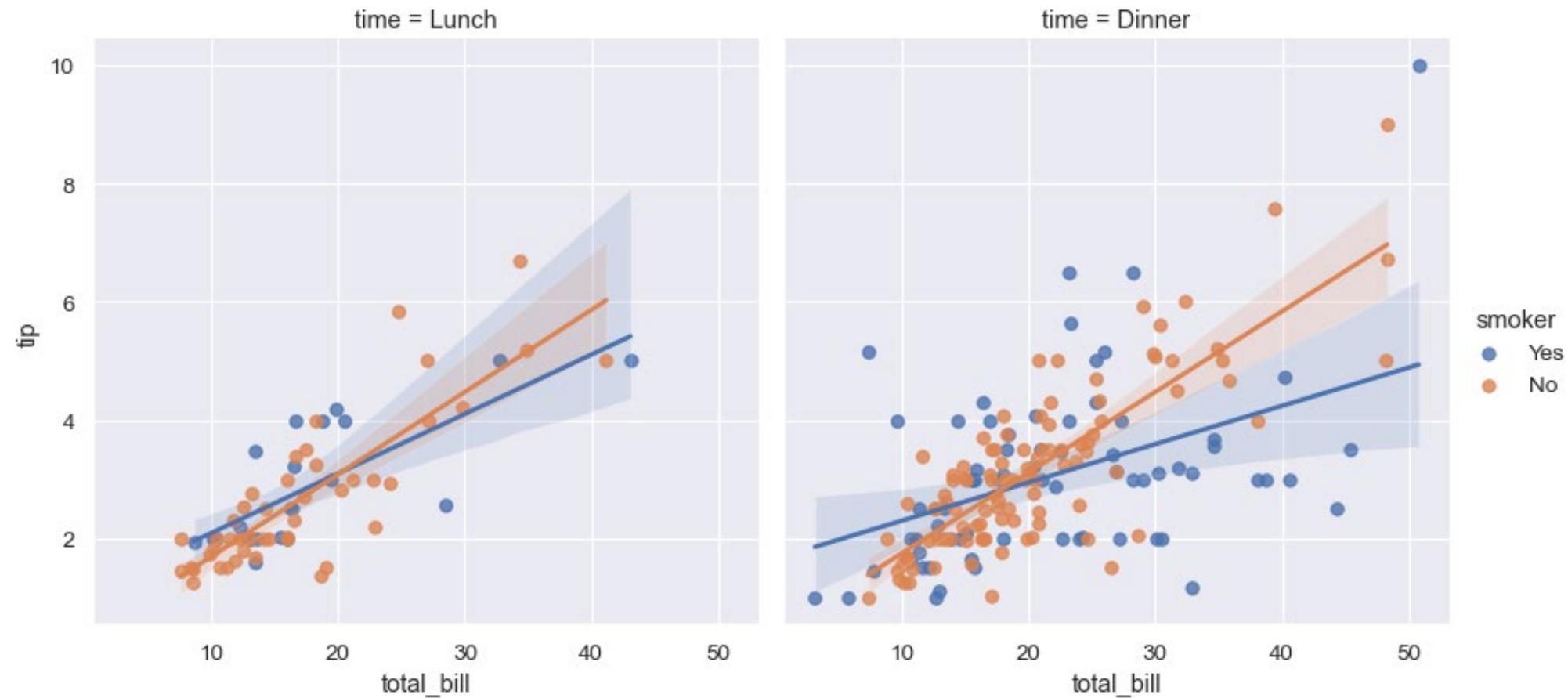
- | • Parameter | Description |
|------------------|---|
| • data=tips | Uses the tips dataset (built-in in Seaborn). |
| • x="total_bill" | X-axis → Total bill amount. |
| • y="tip" | Y-axis → Tip amount. |
| • col="time" | Creates separate plots for Lunch & Dinner. |
| • hue="smoker" | Colors points based on whether the person is a smoker or not. |
| • style="smoker" | Uses different marker shapes for smokers/non-smokers. |
| • size="size" | Adjusts point size based on the size of the dining party. |

Statistical estimations

- `fmri = sns.load_dataset("fmri")`
- `sns.relplot(data=fmri, kind="line", x="timepoint", y="signal", col="region", hue="event", style="event")`



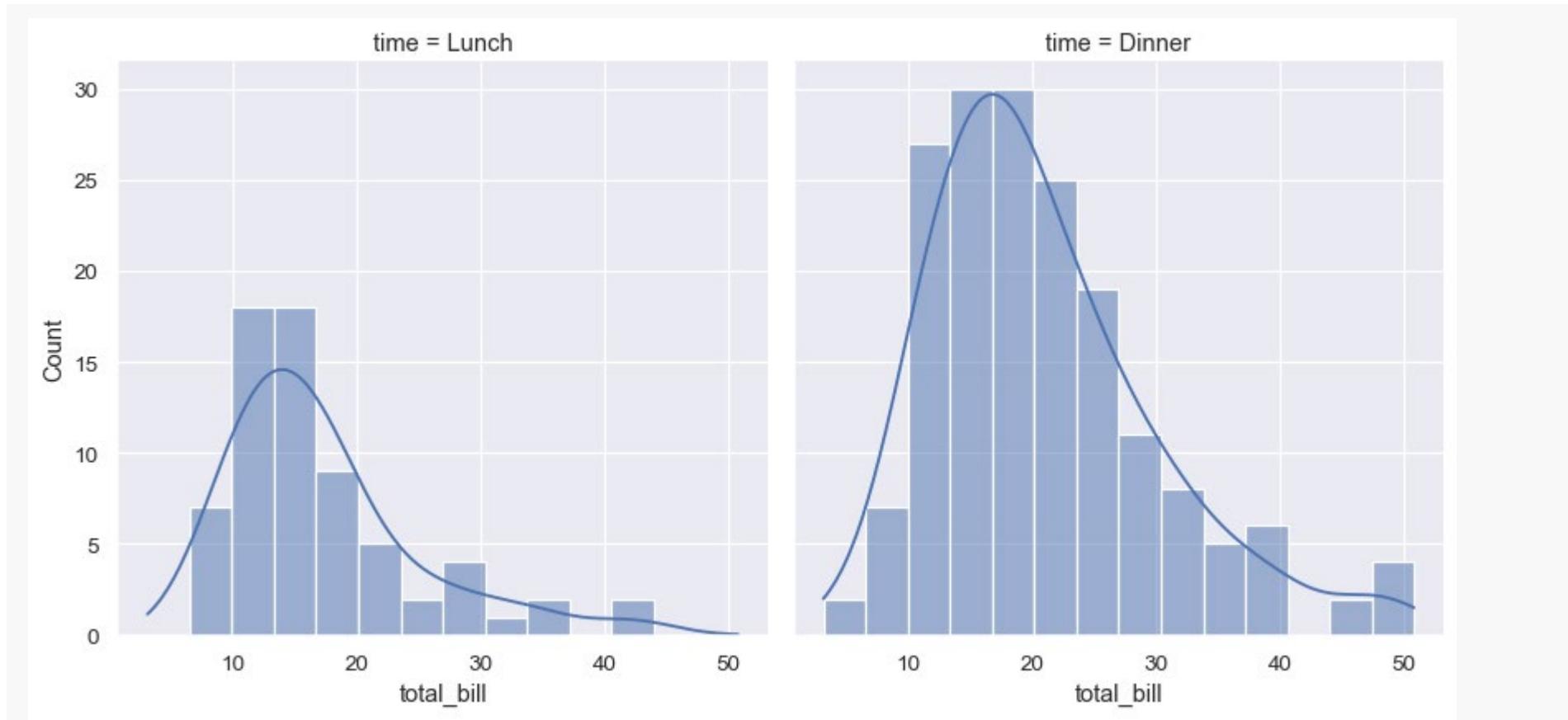
```
sns.lmplot(data=tips, x="total_bill", y="tip", col="time", hue="smoker")
```



Distributional representations

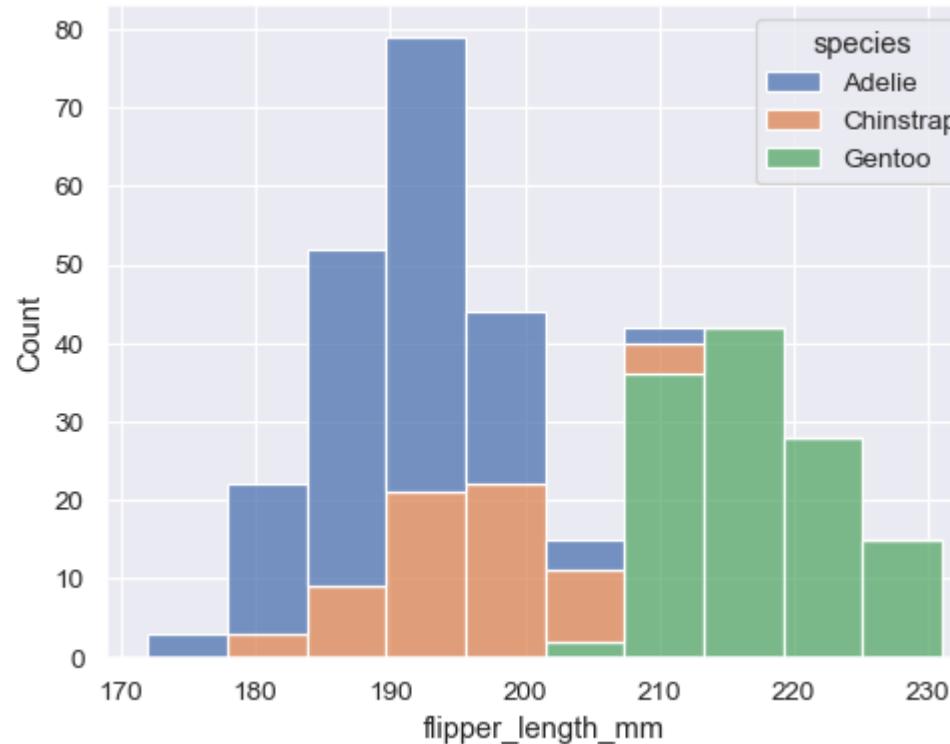
Statistical analyses require knowledge about the distribution of variables in your dataset. The seaborn function `displot()` supports several approaches to visualizing distributions.

```
sns.displot(data=tips, x="total_bill", col="time", kde=True)
```



Sns.histplot()

- penguins = sns.load_dataset("penguins")
- sns.histplot(data=penguins, x="flipper_length_mm", hue="species", multiple="stack")



Kernel density plot

- `sns.kdeplot(data=penguins, x="flipper_length_mm", hue="species", multiple="stack")`

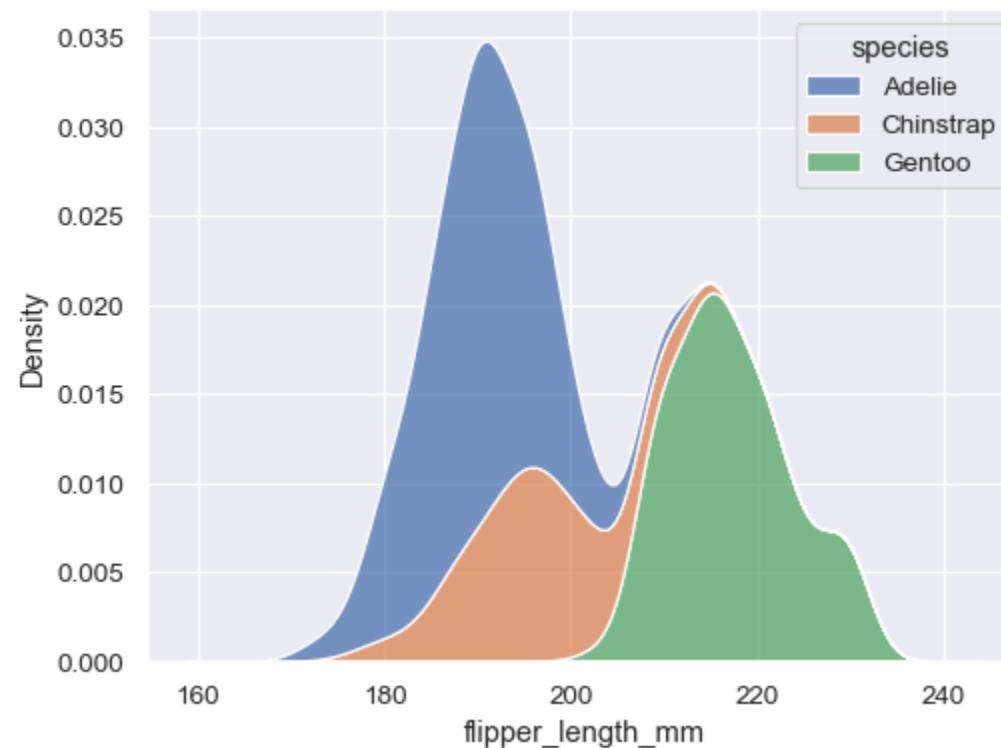


Figure-level vs. axes-level functions

- Axes-Level Functions
- These functions work at the Matplotlib Axes level. They return a single Axes object, allowing direct Matplotlib customization.
 - `sns.scatterplot()`
 - `sns.lineplot()`
 - `sns.histplot()`
 - `sns.boxplot()`
- Figure-Level Functions
- These functions create an entire figure (not just one Axes). They use Seaborn's built-in FacetGrid, which allows multi-plot visualization.
 - `sns.relplot()`
 - `sns.catplot()`
 - `sns.displot()`
 - `sns.pairplot()`

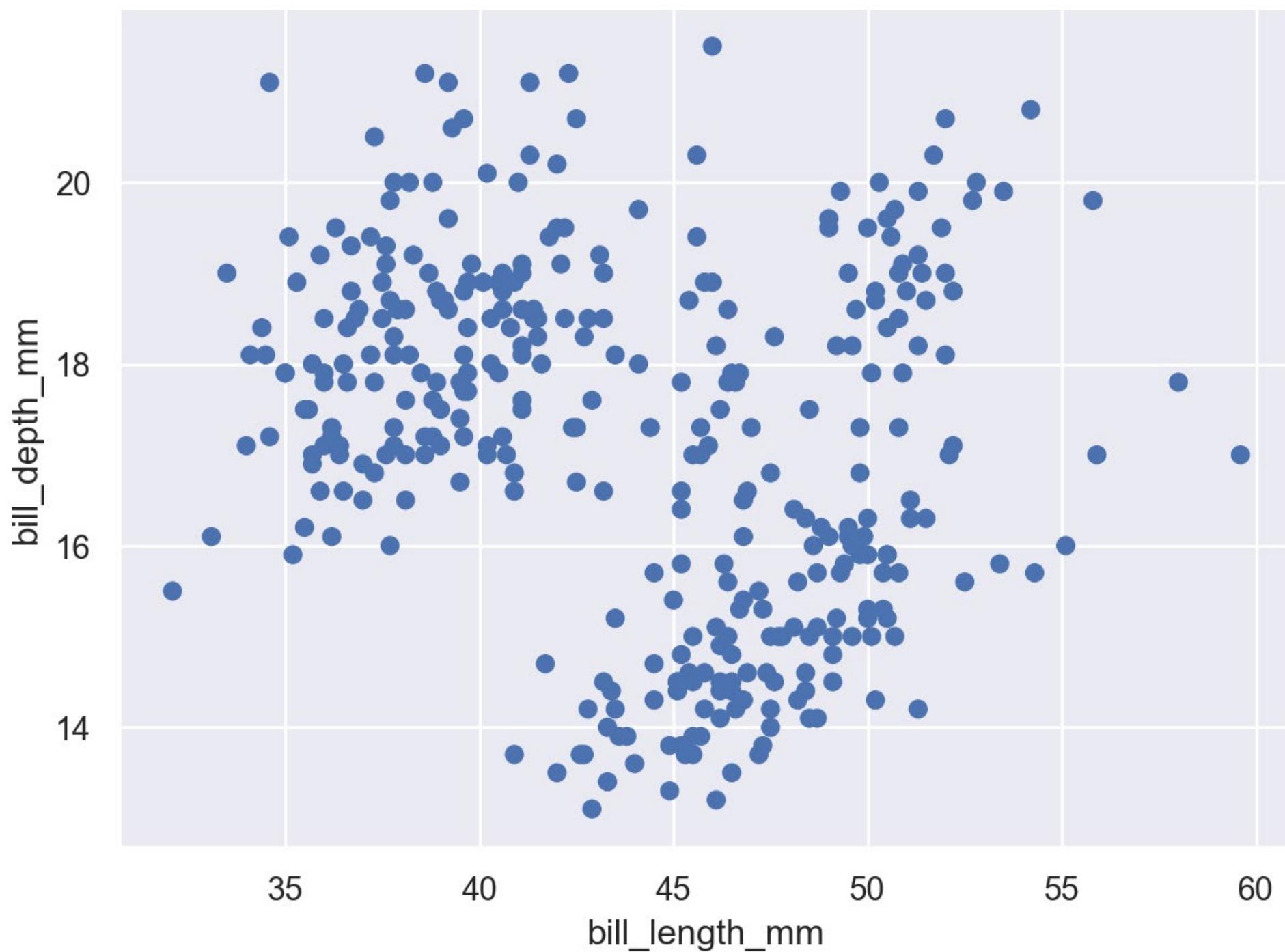
Seaborn: Accepted Data Structures

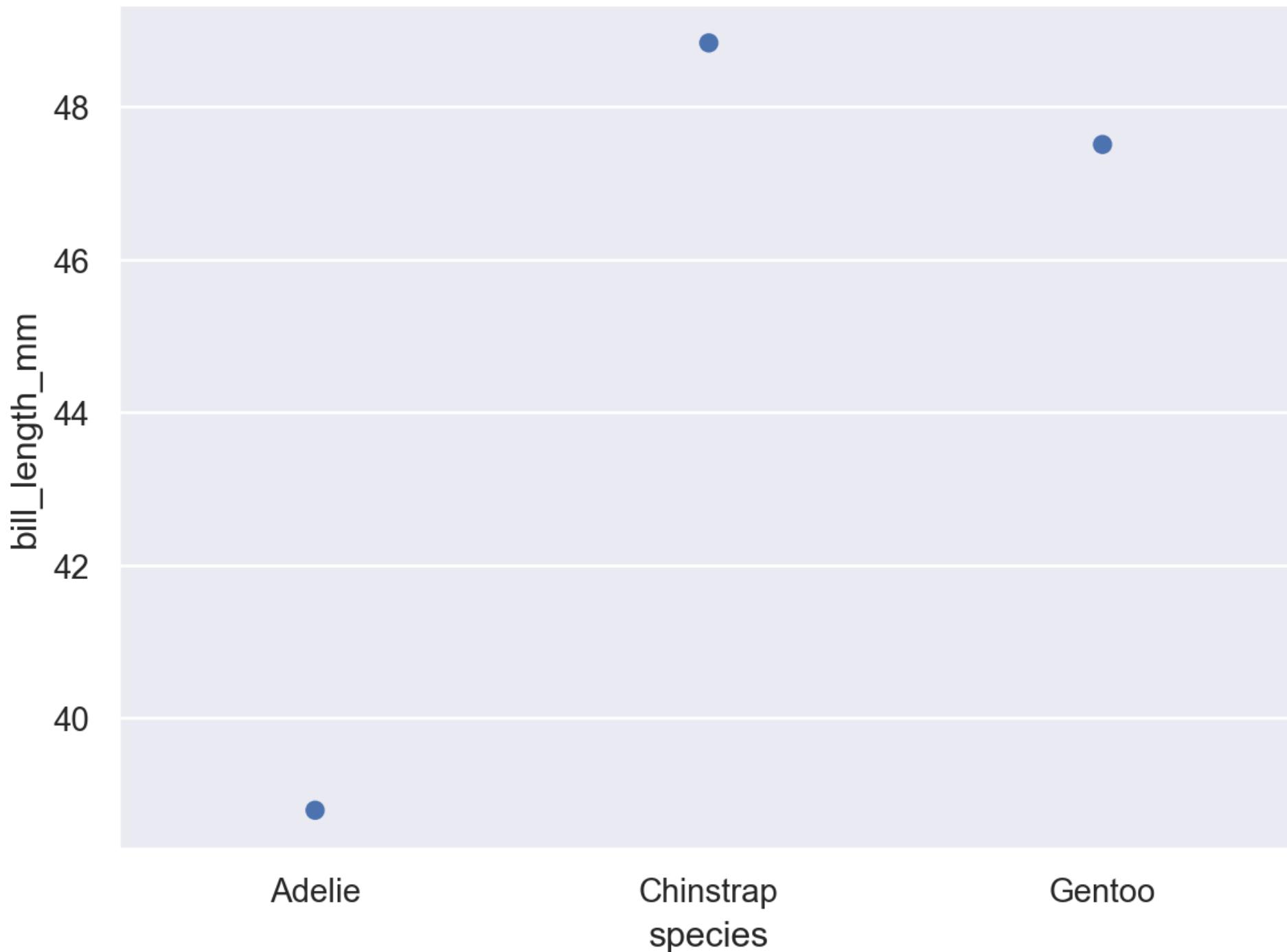
- Seaborn supports multiple dataset formats, mainly from **pandas**, **numpy**, and **Python built-ins** (lists, dicts). Different structures impact how plots are created.

Format	Long-Form (Tidy Data)	Wide-Form
Structure	Each variable has its own column	Variables spread across multiple columns
Example	flights dataset (year, month, passengers)	Pivoted flights_wide (each month as a column)
Usage	More flexible, explicit variable assignment	Easier for simple plots
Customization	More intuitive for complex plots	Requires transformations for some visualizations

Object-Oriented API in seaborn

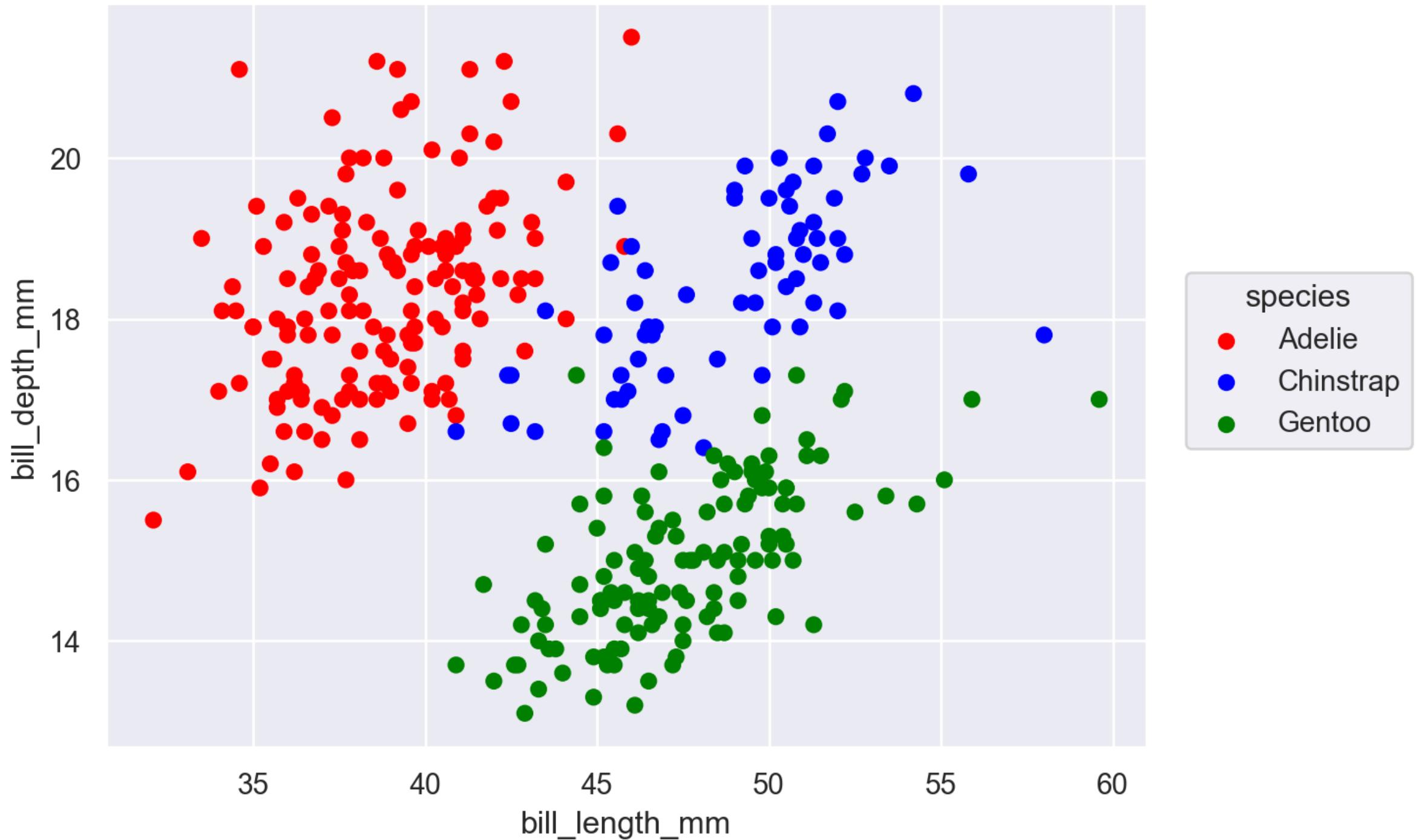
- The `seaborn.objects` module introduces an Object-Oriented API (OO API) for building visualizations. Instead of using function-based methods like `sns.scatterplot()`, you create a `Plot` object and build the visualization step-by-step using method chaining.
- `import seaborn.objects as so`
- `import seaborn as sns`
- `penguins = sns.load_dataset("penguins")`
- `so.Plot(penguins,x="bill_length_mm",y="bill_depth_mm").add(so.Dot())`





Customizing Colors & Aesthetics

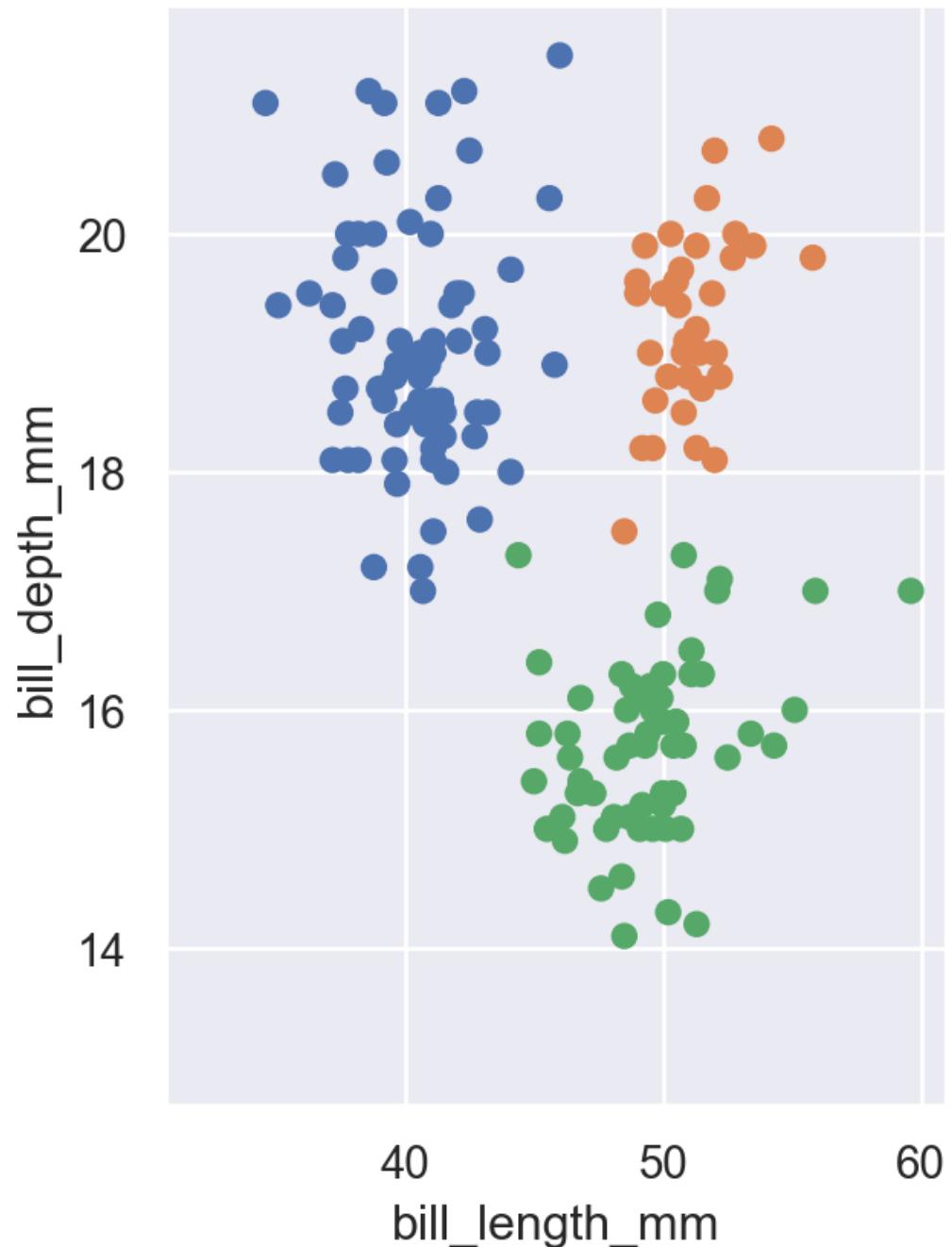
- Instead of hue= in function-based Seaborn, you map aesthetics explicitly.
- `so.Plot(penguins, x="bill_length_mm", y="bill_depth_mm", color="species").add(so.Dot()).scale(color={"Adelie": "red", "Chinstrap": "blue", "Gentoo": "green"})`



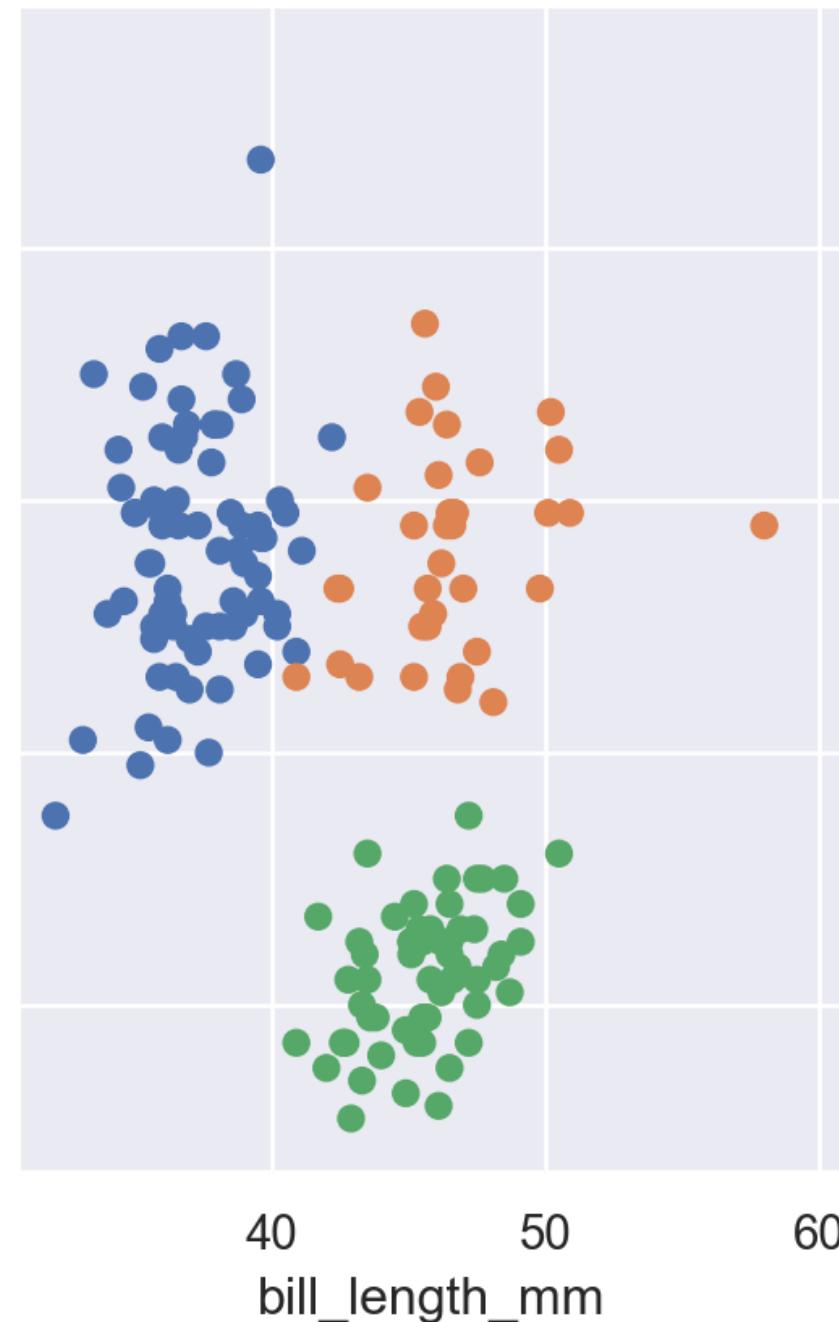
Faceting: Splitting into Multiple Plots

- Instead of sns.FacetGrid(), you use .facet()
- so.Plot(penguins, x="bill_length_mm", y="bill_depth_mm", color="species").facet(col="sex").add(so.Dot())

Male



Female



species

- Adelie
- Chinstrap
- Gentoo

Available markers

Function-based Seaborn

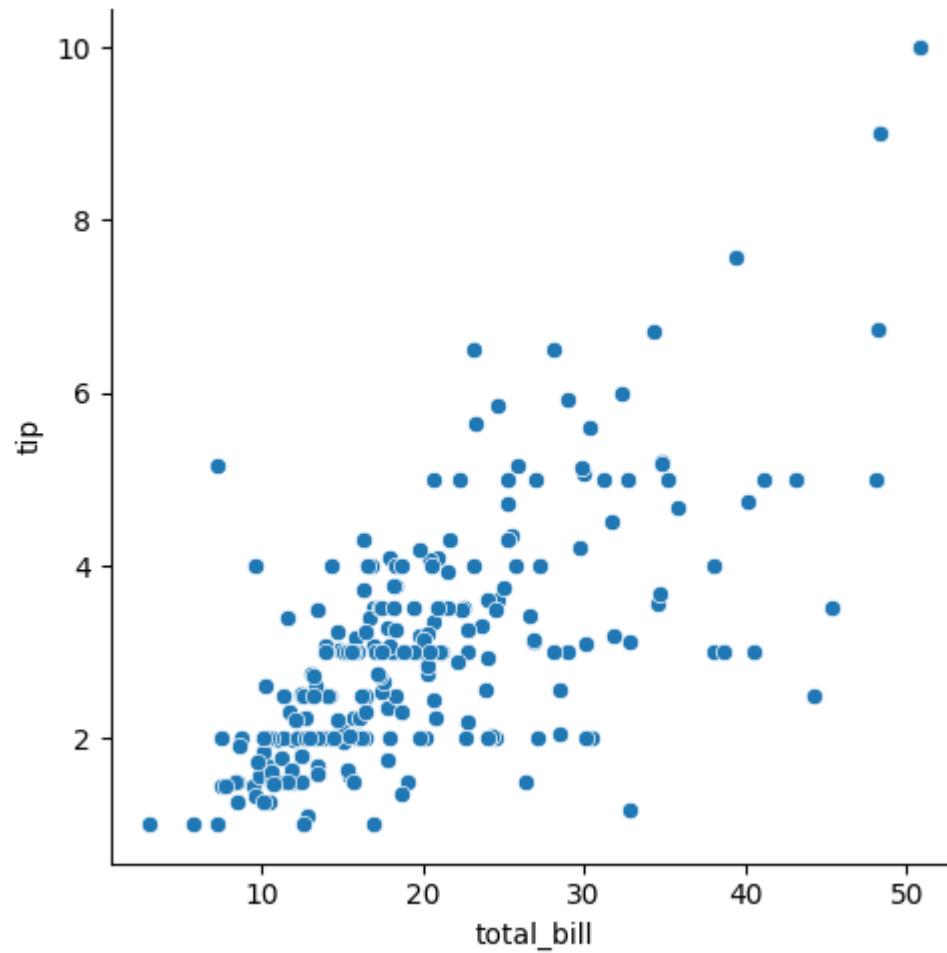
`sns.scatterplot()`
`sns.histplot()`
`sns.lineplot()`
`sns.boxplot()`
`sns.violinplot()`

Object-Oriented Equivalent

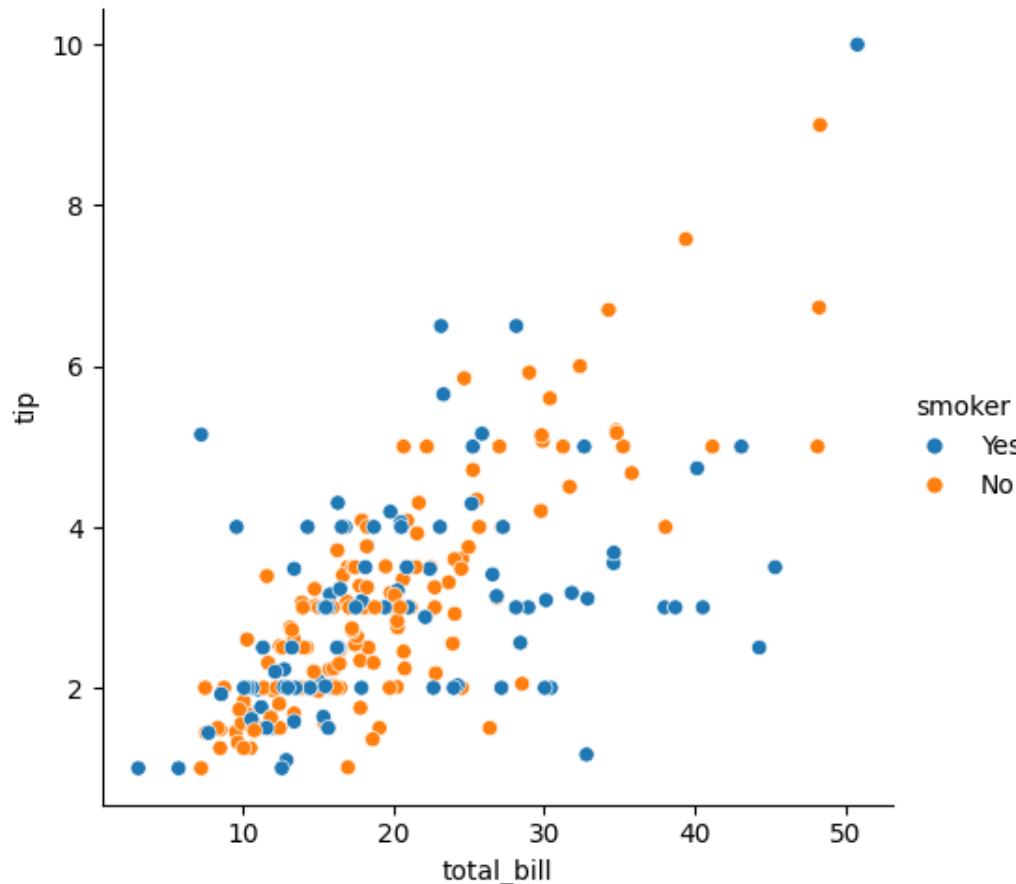
`.add(so.Dot())`
`.add(so.Bars())`
`.add(so.Line())`
`.add(so.Box())`
`.add(so.Violin())`

Visualizing Statistical Relationships in Seaborn

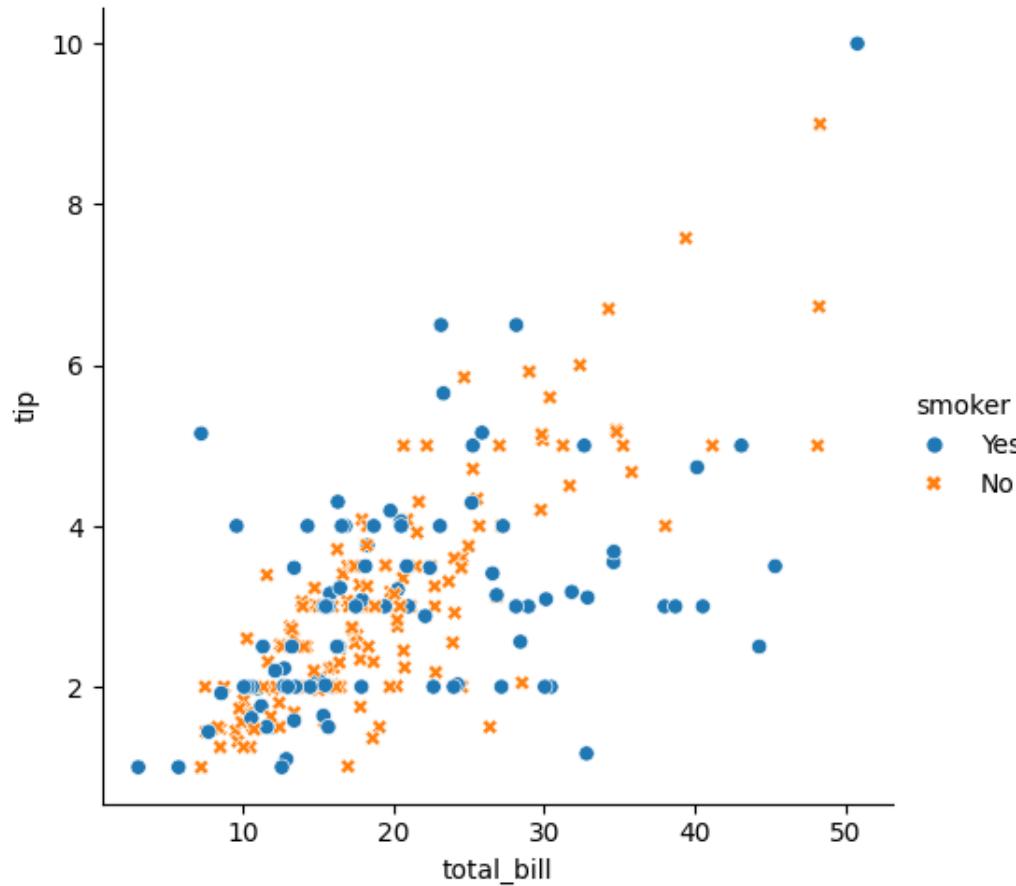
- Seaborn provides powerful tools to visualize statistical relationships between variables using scatter plots and line plots. The key function for this is `sns.relplot()`, which acts as a figure-level function that wraps:
 - `sns.scatterplot()` (when `kind="scatter"`, default)
 - `sns.lineplot()` (when `kind="line"`)
 - `import seaborn as sns`
 - `import matplotlib.pyplot as plt`
 - `# Load dataset`
 - `tips = sns.load_dataset("tips")`
 - `# Basic scatter plot`
 - `sns.relplot(data=tips, x="total_bill", y="tip")`
 - `plt.show()`



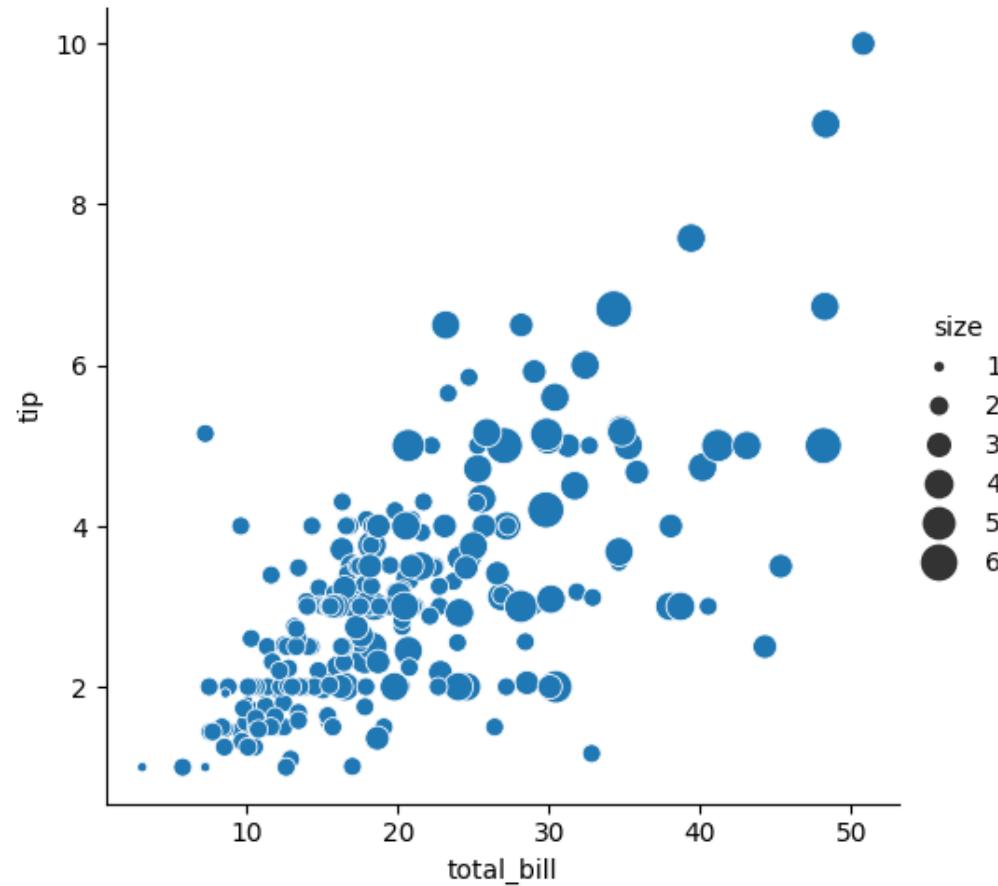
```
sns.relplot(data=tips, x="total_bill", y="tip",  
hue="smoker")
```



```
sns.relplot(data=tips, x="total_bill", y="tip",  
hue="smoker", style="smoker")
```

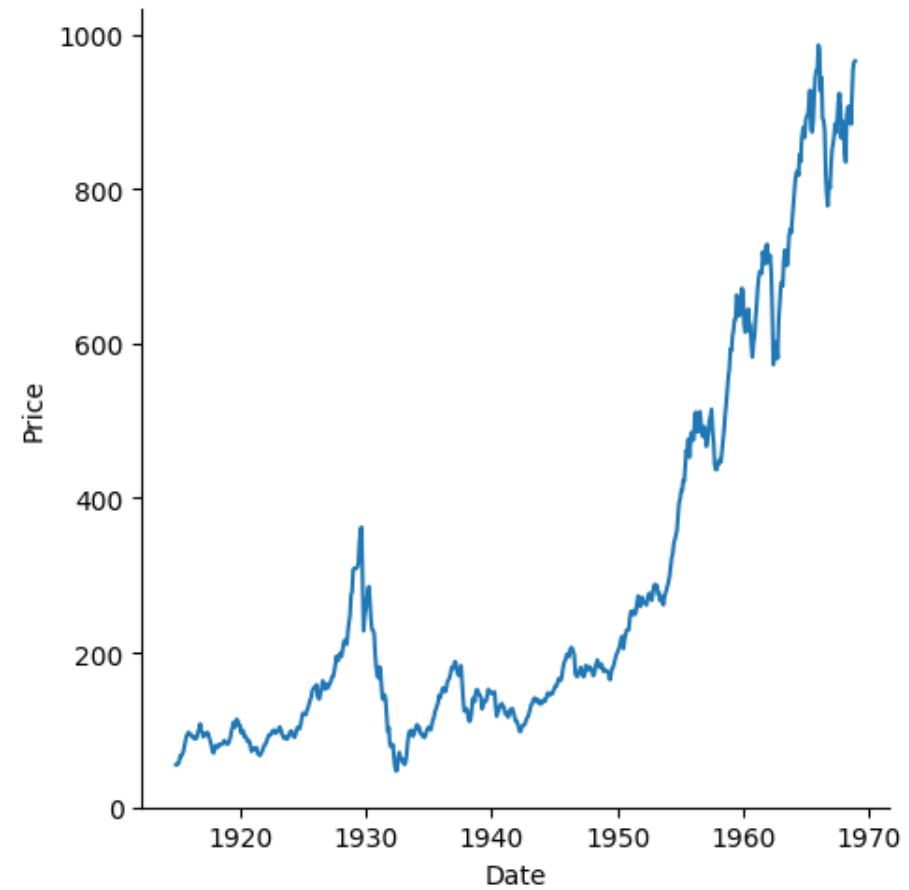


```
sns.relplot(data=tips, x="total_bill", y="tip",  
size="size", sizes=(15, 200))
```



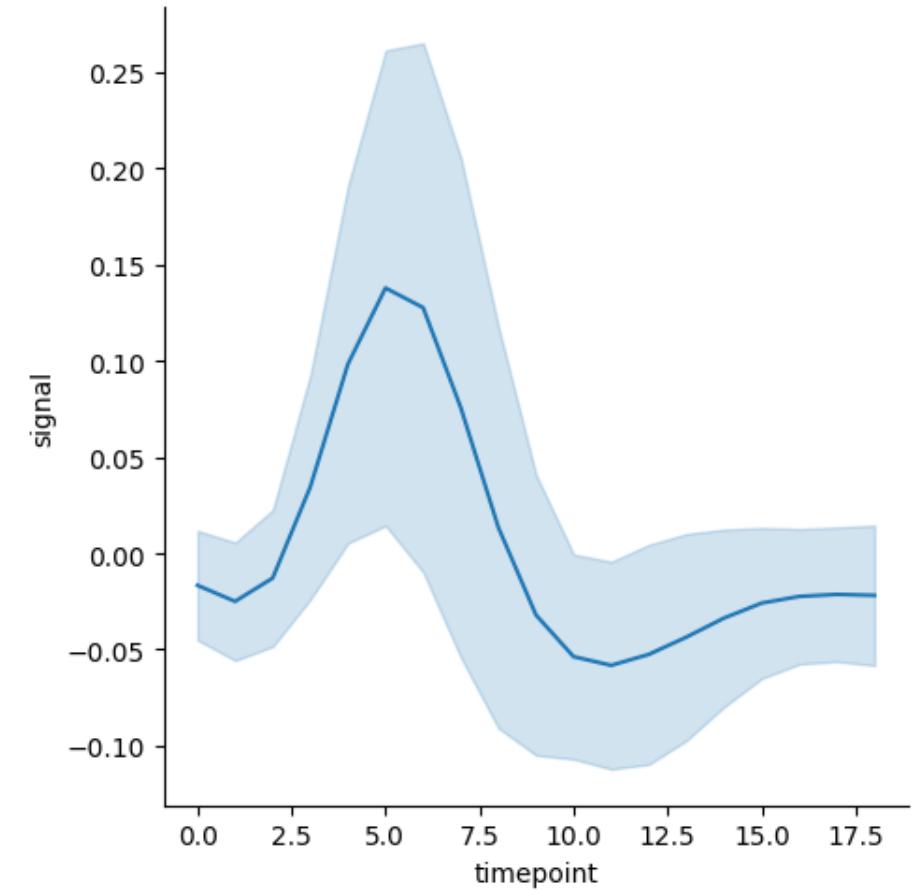
Line plot – Dow jones

```
sns.relplot(data=dowjones, x="Date", y="Price", kind="line")
```



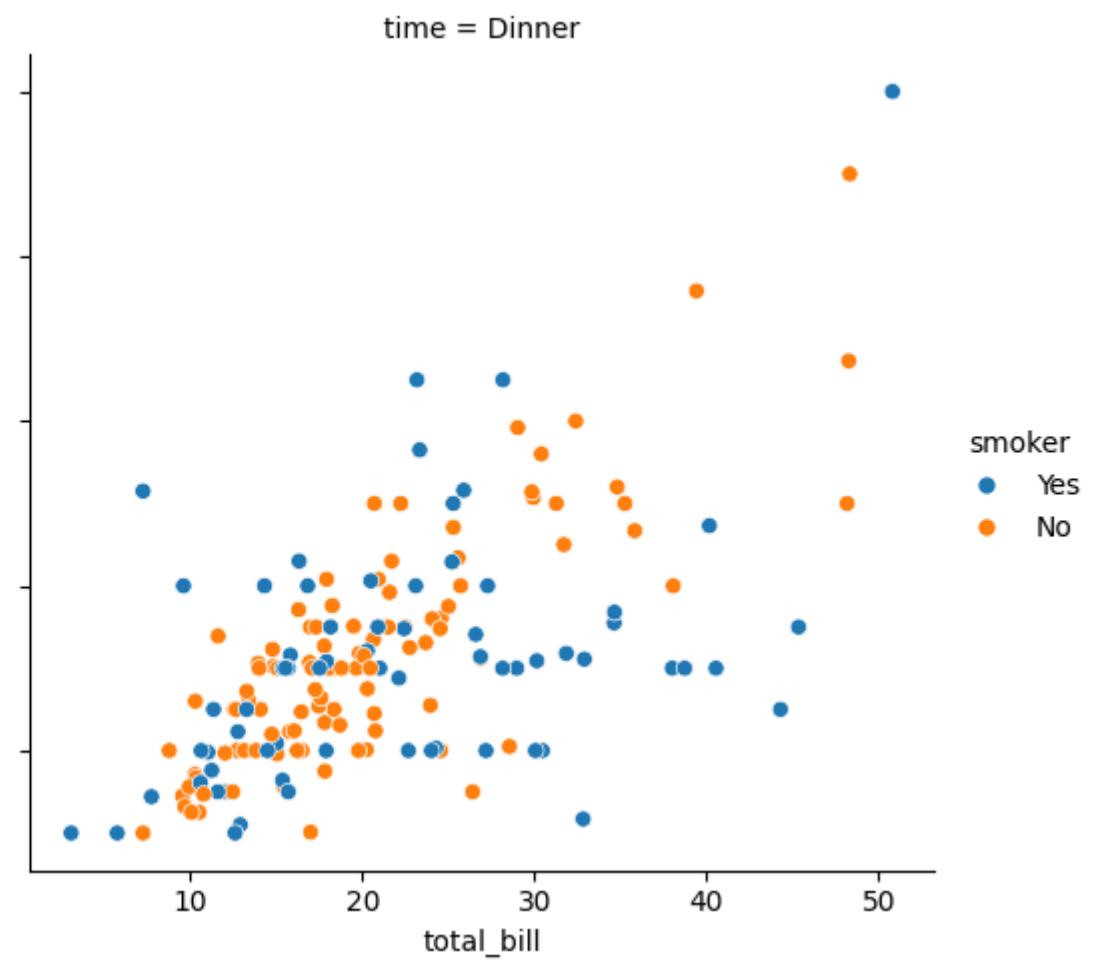
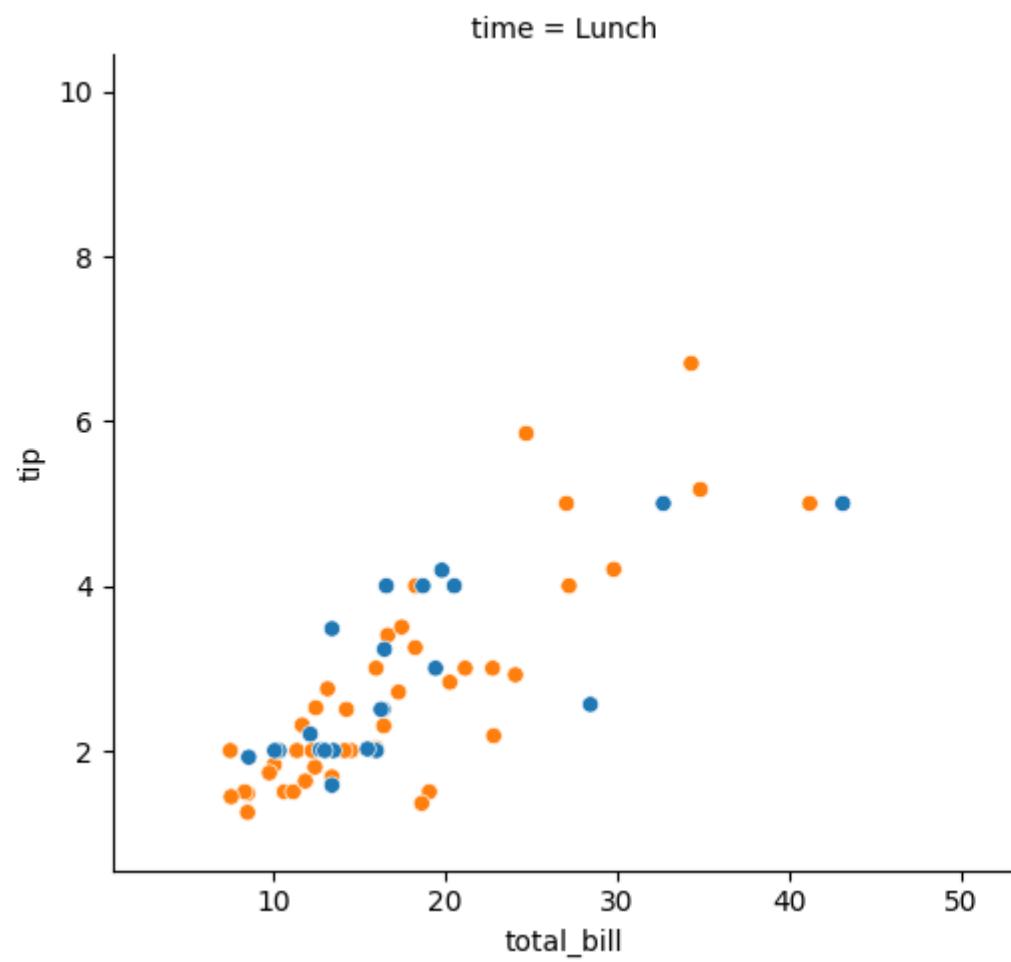
Adding Confidence Intervals

```
fmri = sns.load_dataset("fmri")
sns.relplot(data=fmri, x="timepoint", y="signal",
kind="line", errorbar="sd")
```

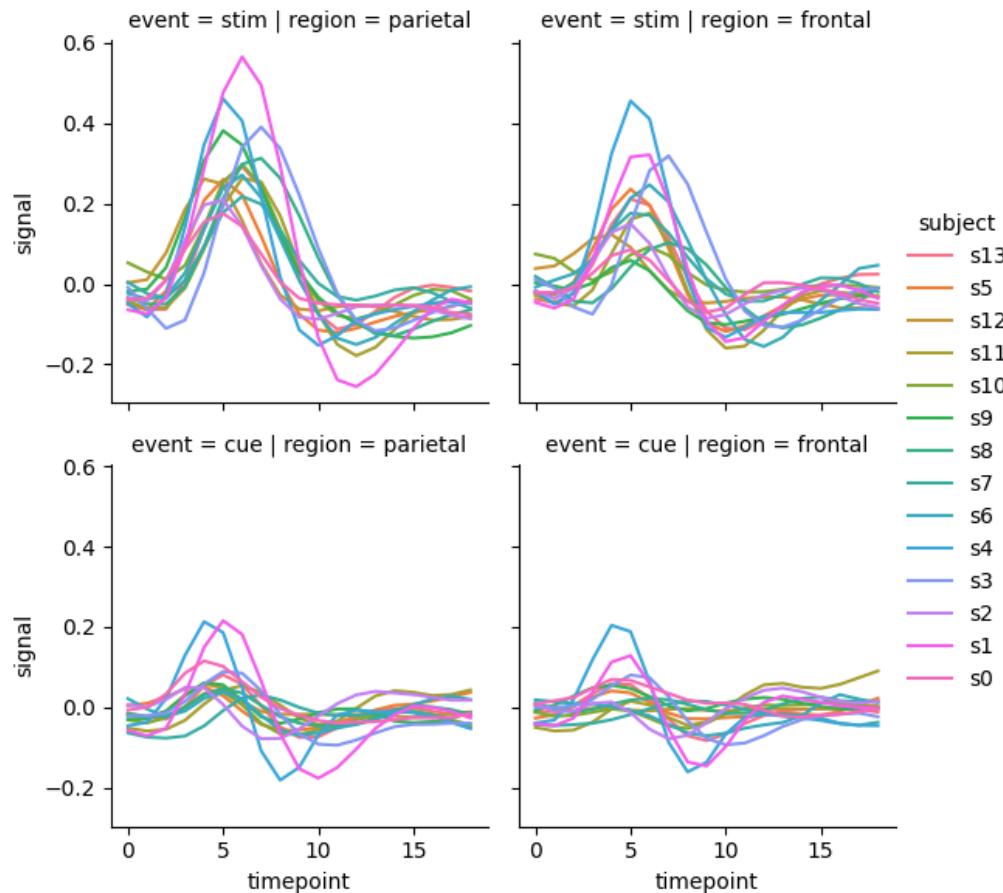


Multi-Plot (Facet) Approach

- Instead of using **hue**, **size**, or **style**, we can **split** data into **subplots**.
 - `sns.relplot(data=tips, x="total_bill", y="tip", hue="smoker", col="time")`

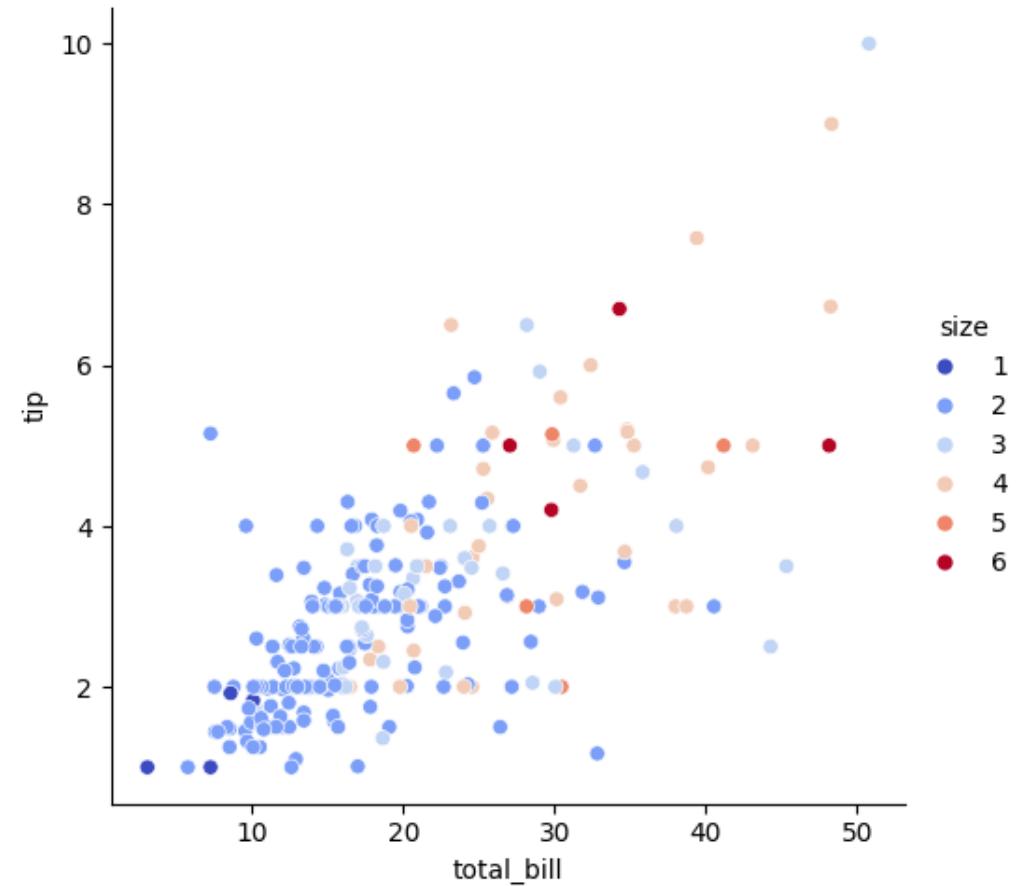


```
sns.relplot(data=fmri, kind="line", x="timepoint", y="signal",
hue="subject", col="region", row="event", height=3,
estimator=None)
```



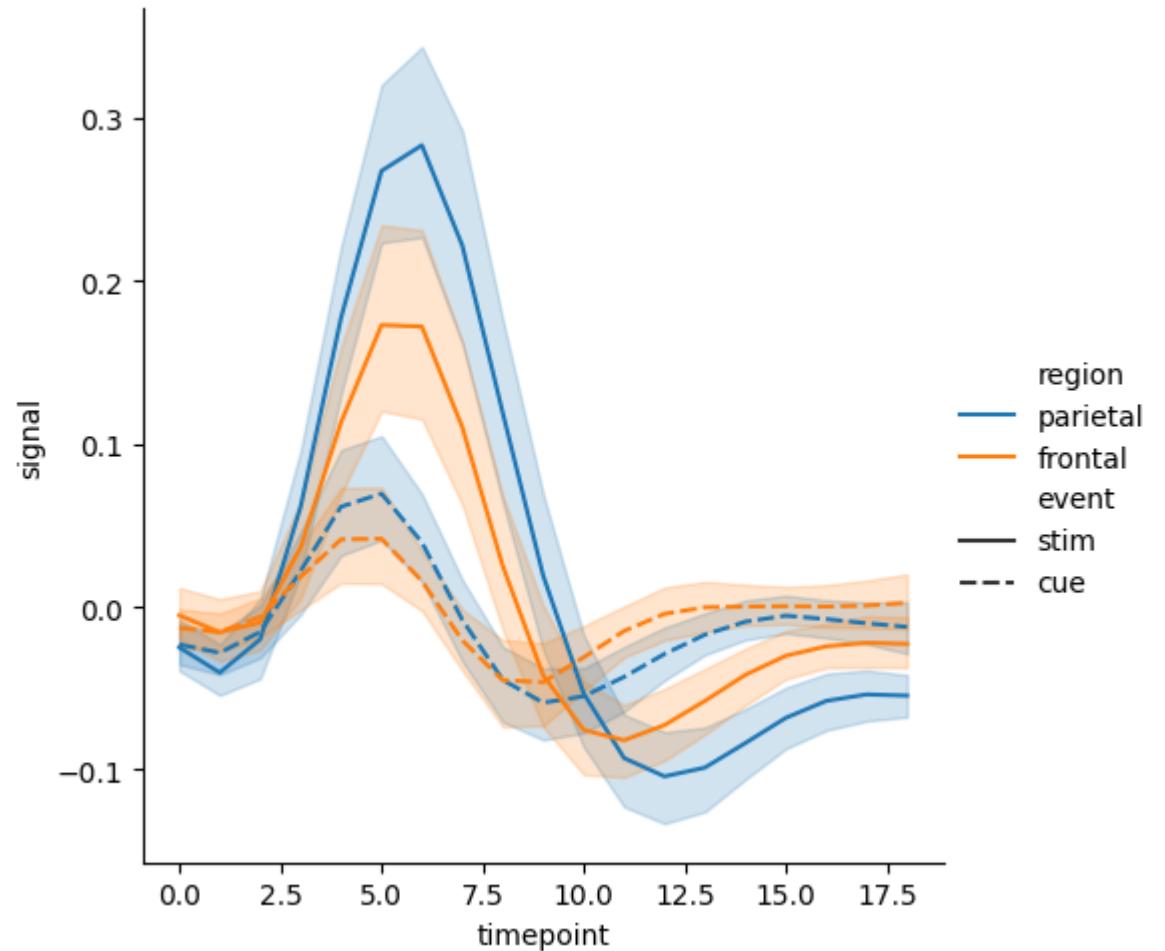
Customising colors

```
sns.relplot(data=tips, x="total_bill", y="tip",
hue="size", palette="coolwarm")
```

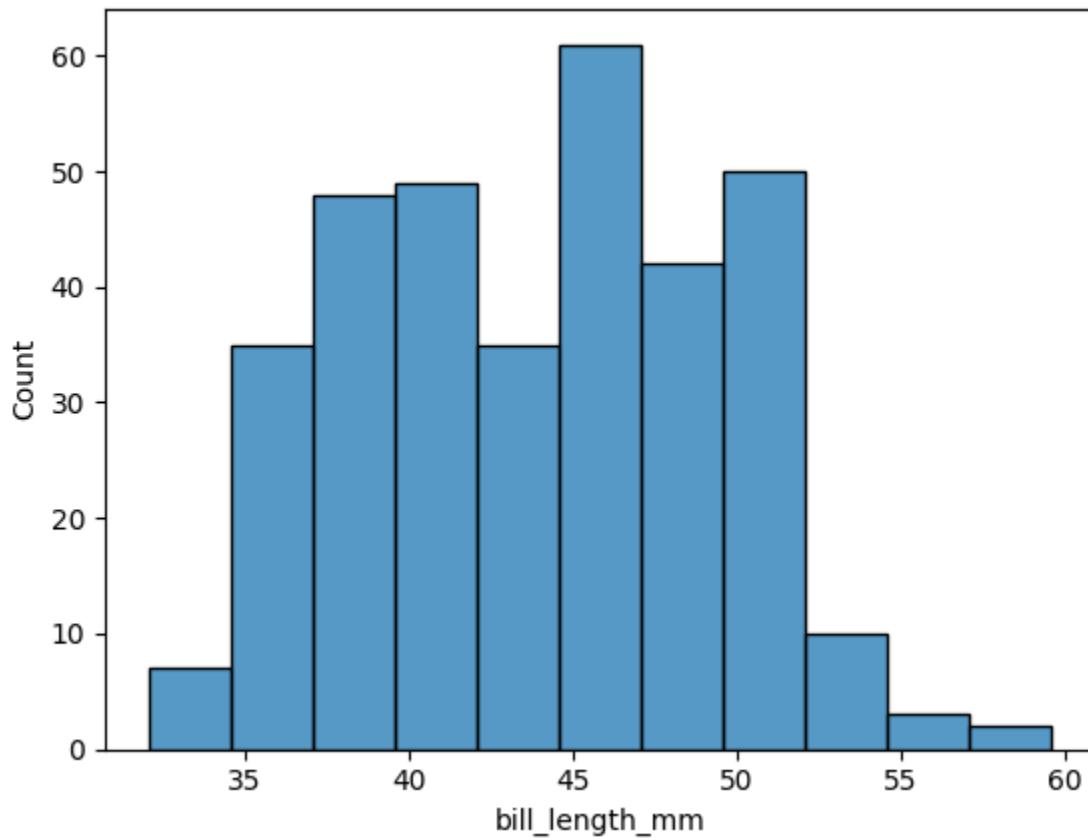


Change Line Style (style)

- `sns.relplot(data=fmri, kind="line", x="timepoint", y="signal", hue="region", style="event")`

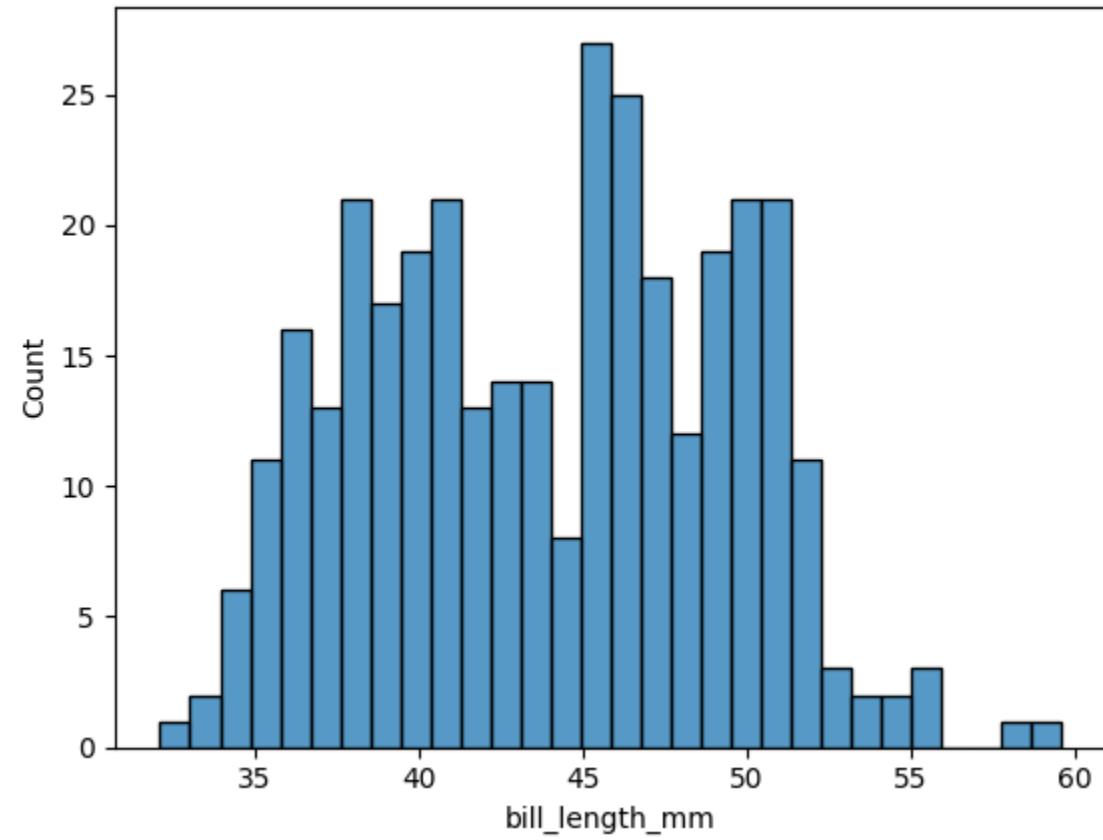


Basic histogram

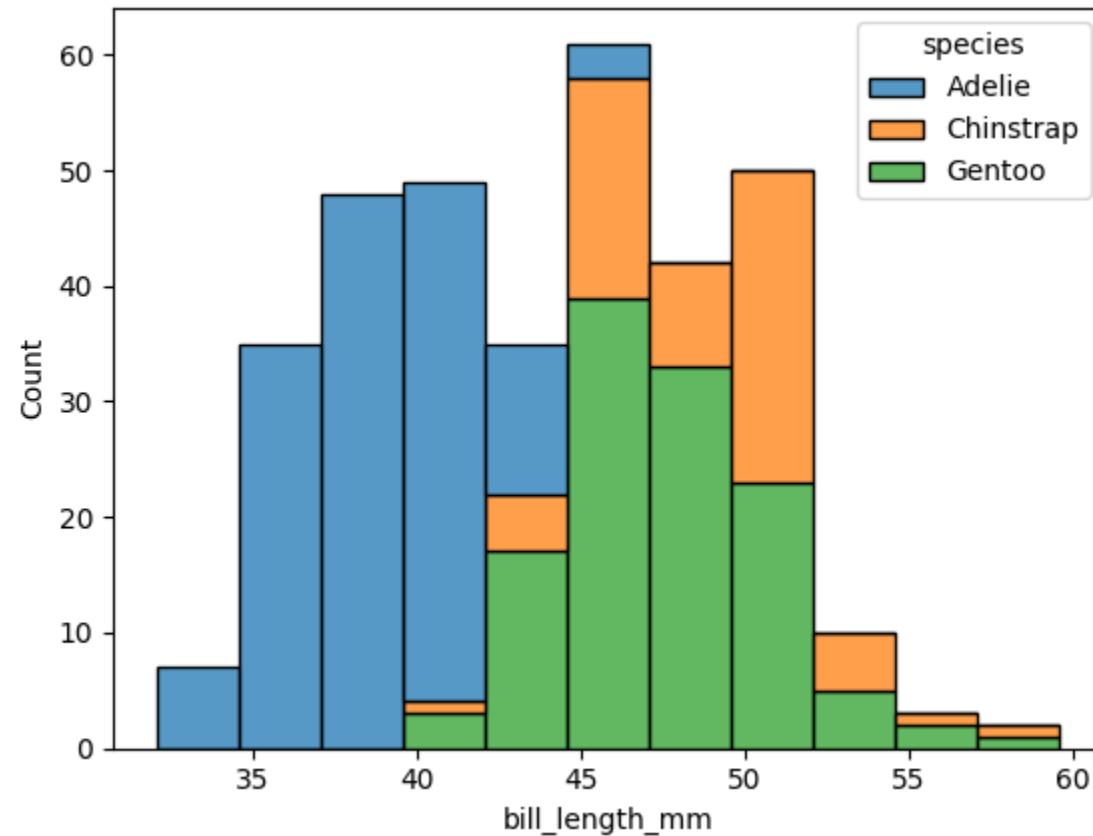


Customizing the Histogram (Adjusting Bin Size)

```
sns.histplot(penguins, x="bill_length_mm", bins=30)
```

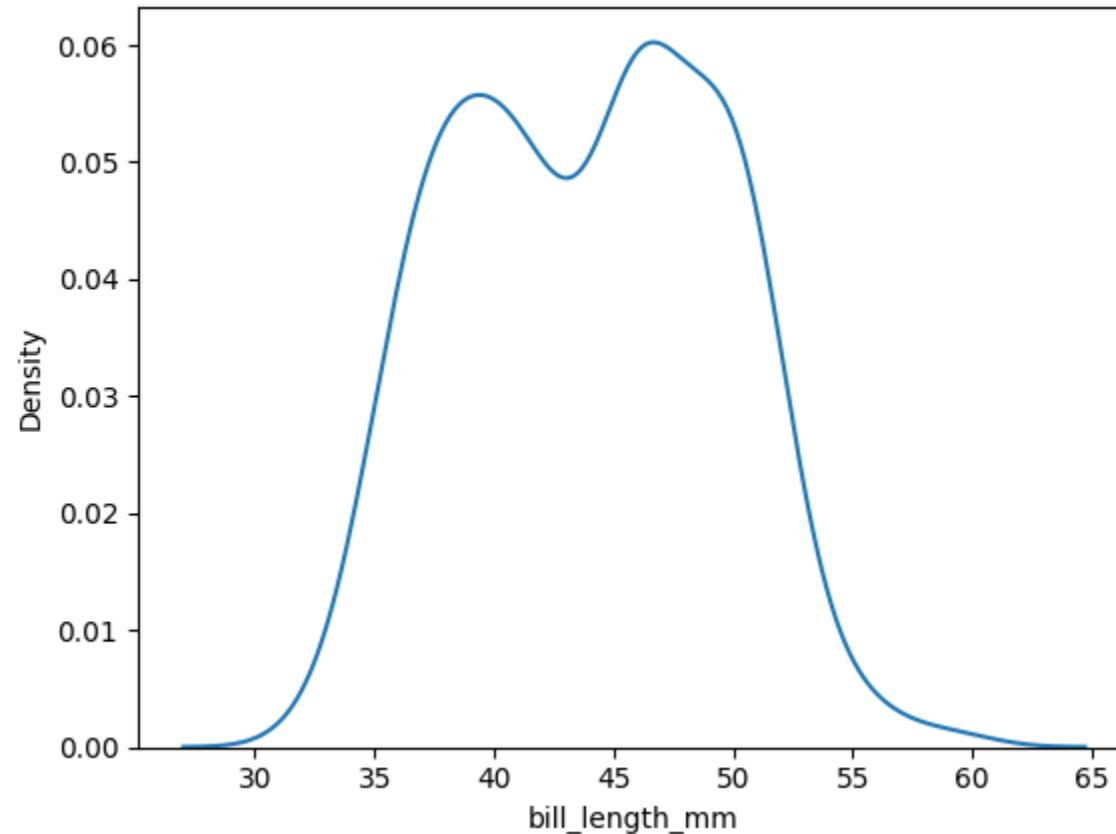


```
sns.histplot(penguins, x="bill_length_mm",  
hue="species", multiple="stack")
```



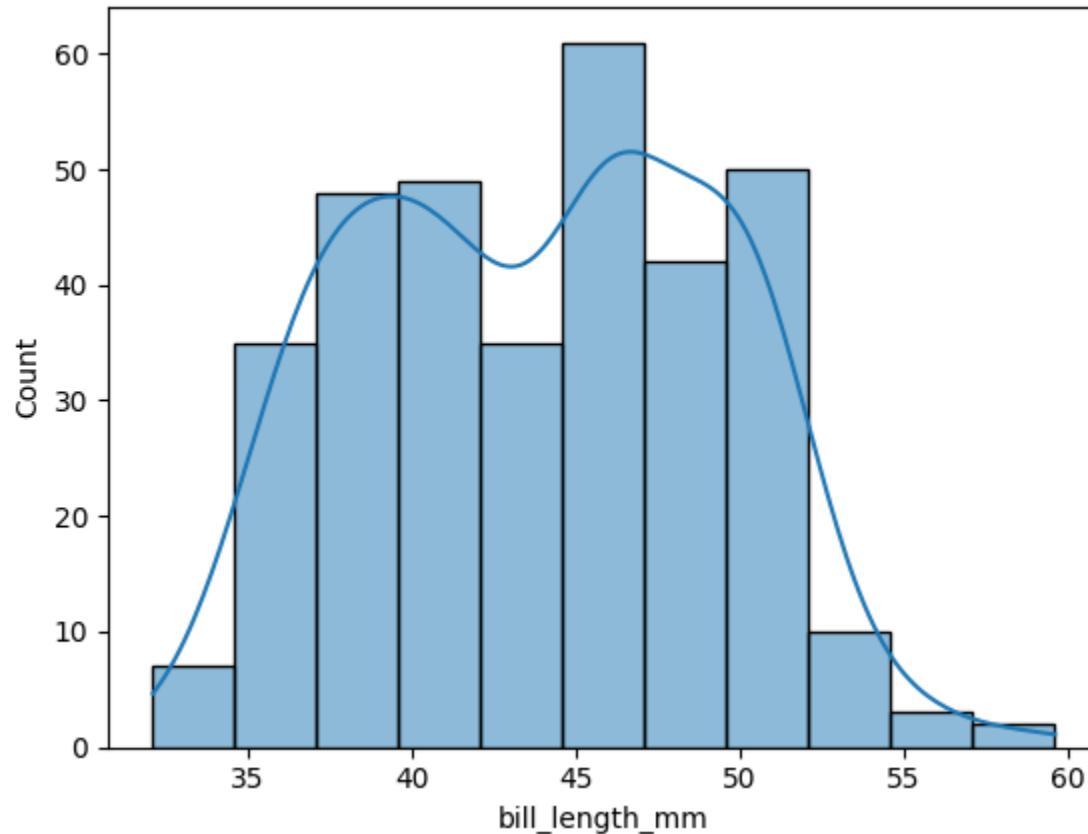
Density plot

```
sns.kdeplot(penguins, x="bill_length_mm")
```



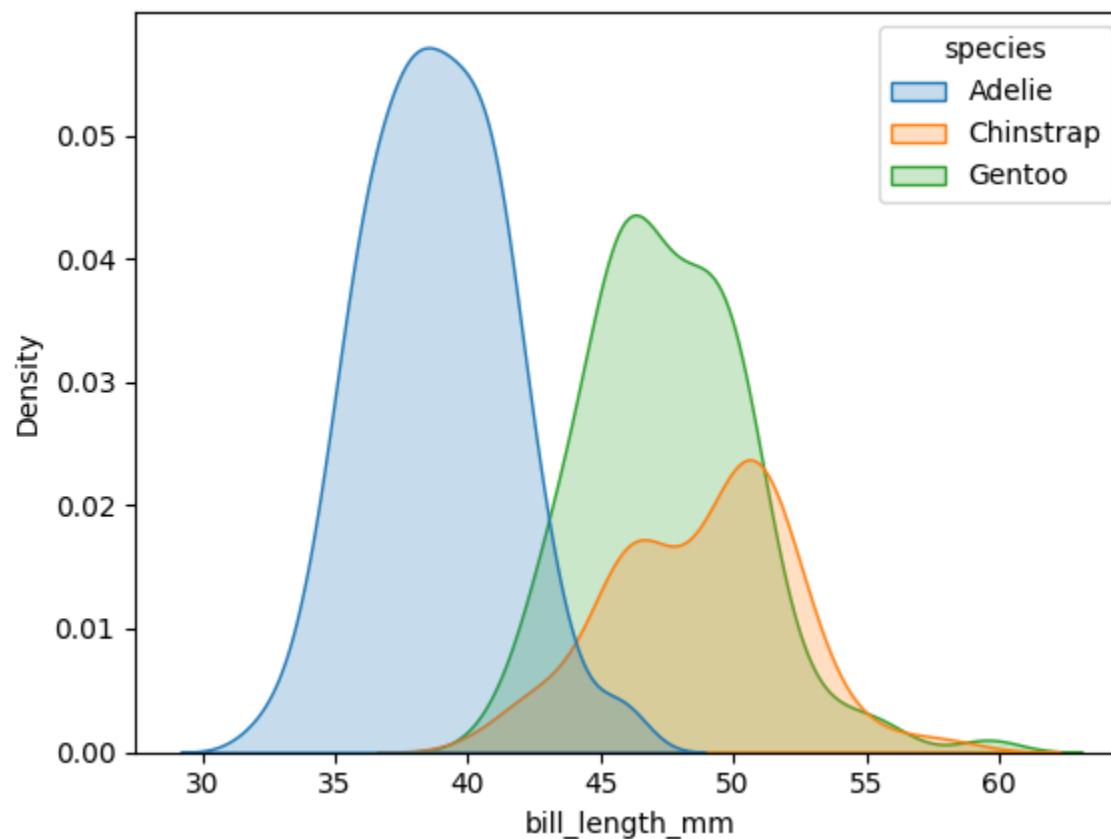
KDE + Histogram Together

```
sns.histplot(penguins, x="bill_length_mm", kde=True)
```



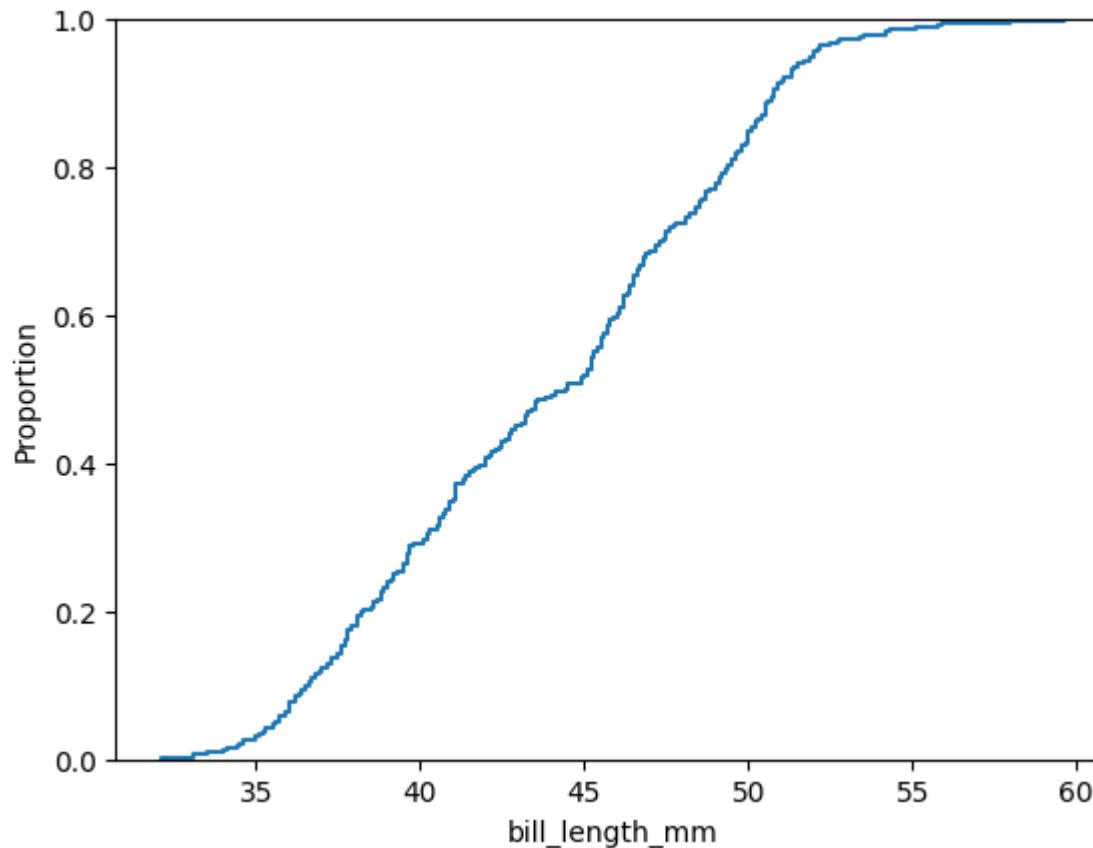
Kde by category

```
sns.kdeplot(penguins, x="bill_length_mm", hue="species", fill=True)
```



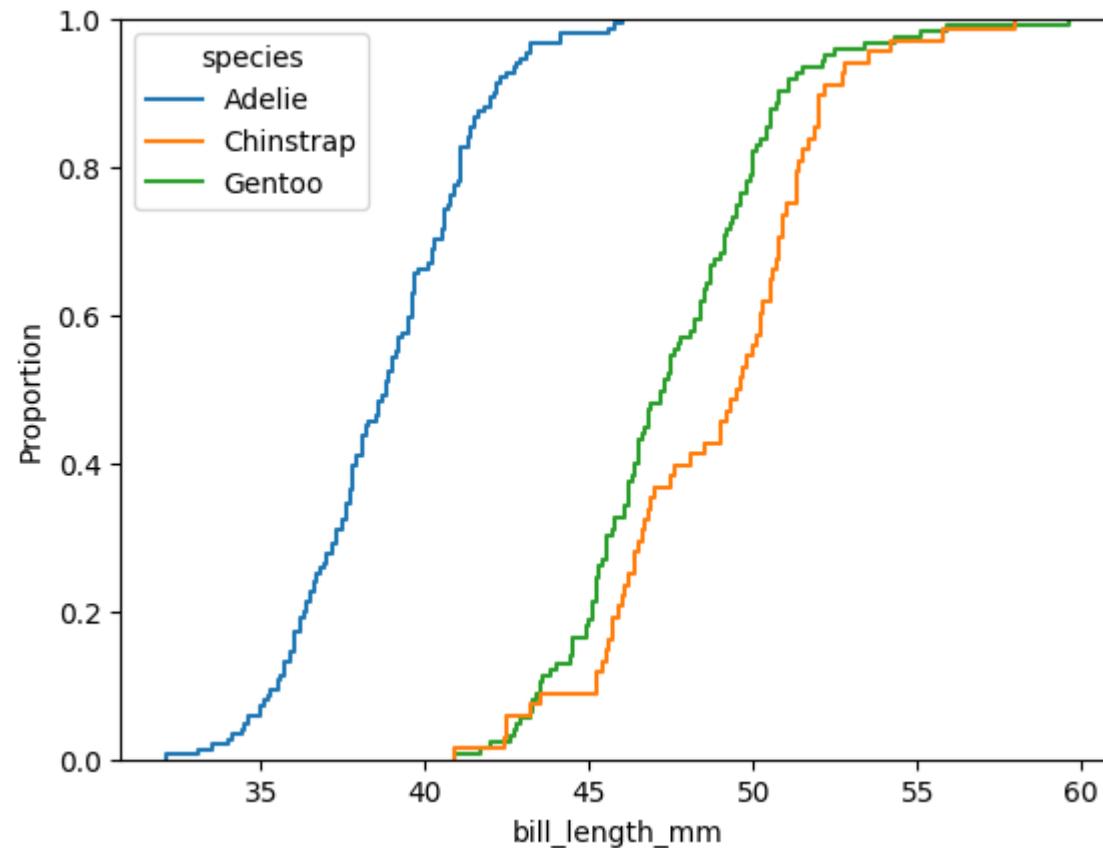
ECDF

```
sns.ecdfplot(penguins, x="bill_length_mm")
```



ECDF with Hue

```
sns.ecdfplot(penguins, x="bill_length_mm", hue="species")
```

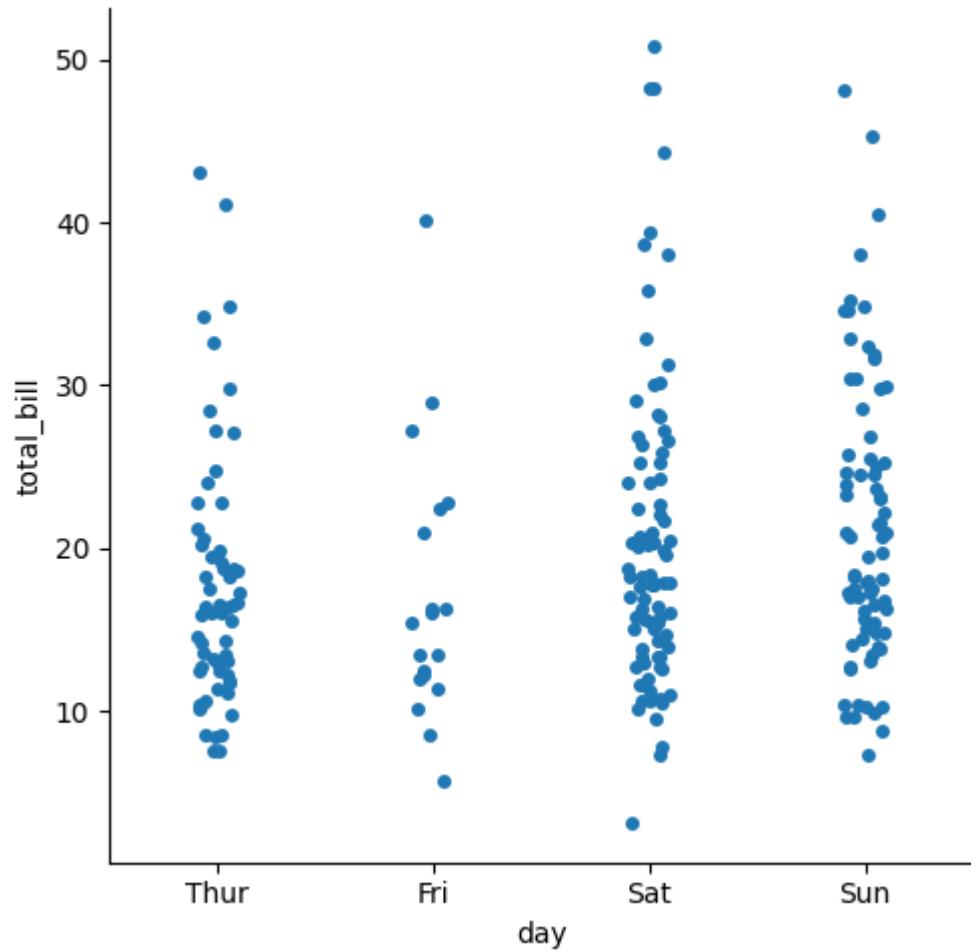


Categorical Plot Types in Seaborn

Category	Plot Type	Function
Scatter-Based	Strip Plot	<code>sns.stripplot()</code>
	Swarm Plot	<code>sns.swarmplot()</code>
Distribution-Based	Box Plot	<code>sns.boxplot()</code>
	Violin Plot	<code>sns.violinplot()</code>
Estimate-Based	Boxen Plot	<code>sns.boxenplot()</code>
	Bar Plot	<code>sns.barplot()</code>
	Count Plot	<code>sns.countplot()</code>
	Point Plot	<code>sns.pointplot()</code>

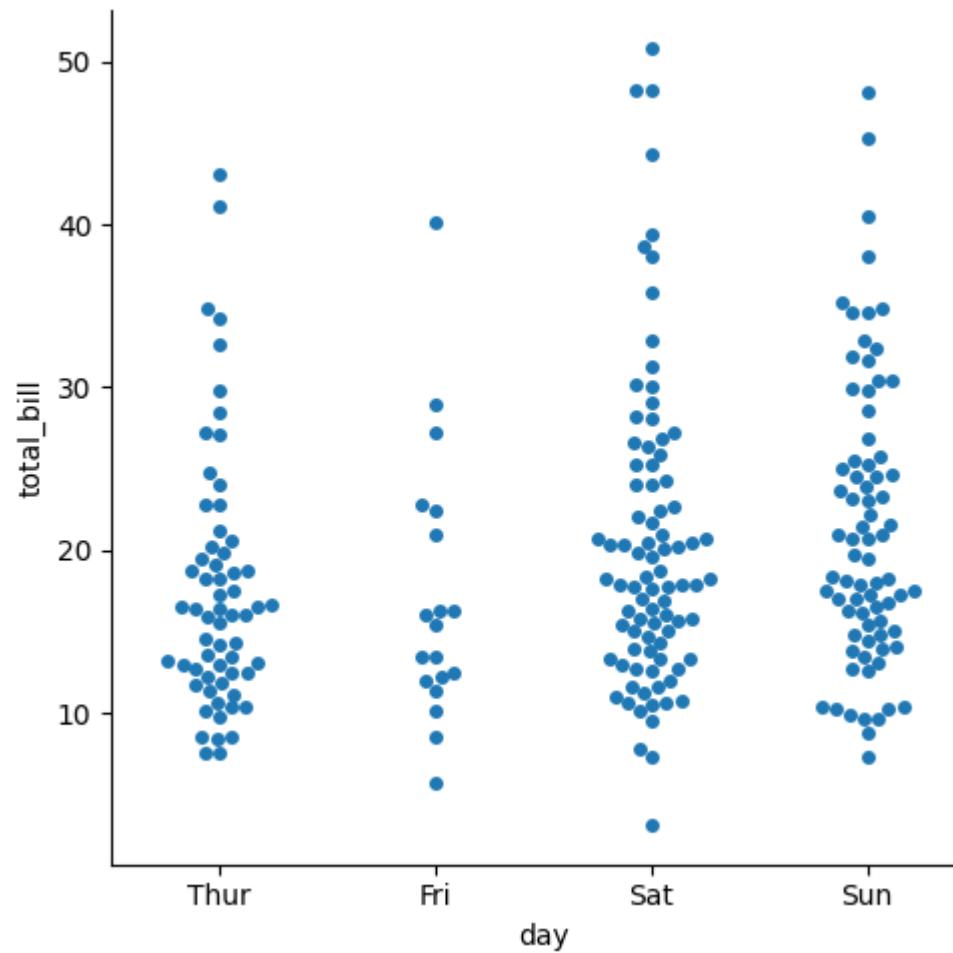
Strip Plot

```
import seaborn as sns  
import matplotlib.pyplot as plt  
  
# Load dataset  
tips = sns.load_dataset("tips")  
  
# Basic strip plot  
sns.catplot(data=tips, x="day", y="total_bill", kind="strip",  
jitter=True)  
  
plt.show()
```



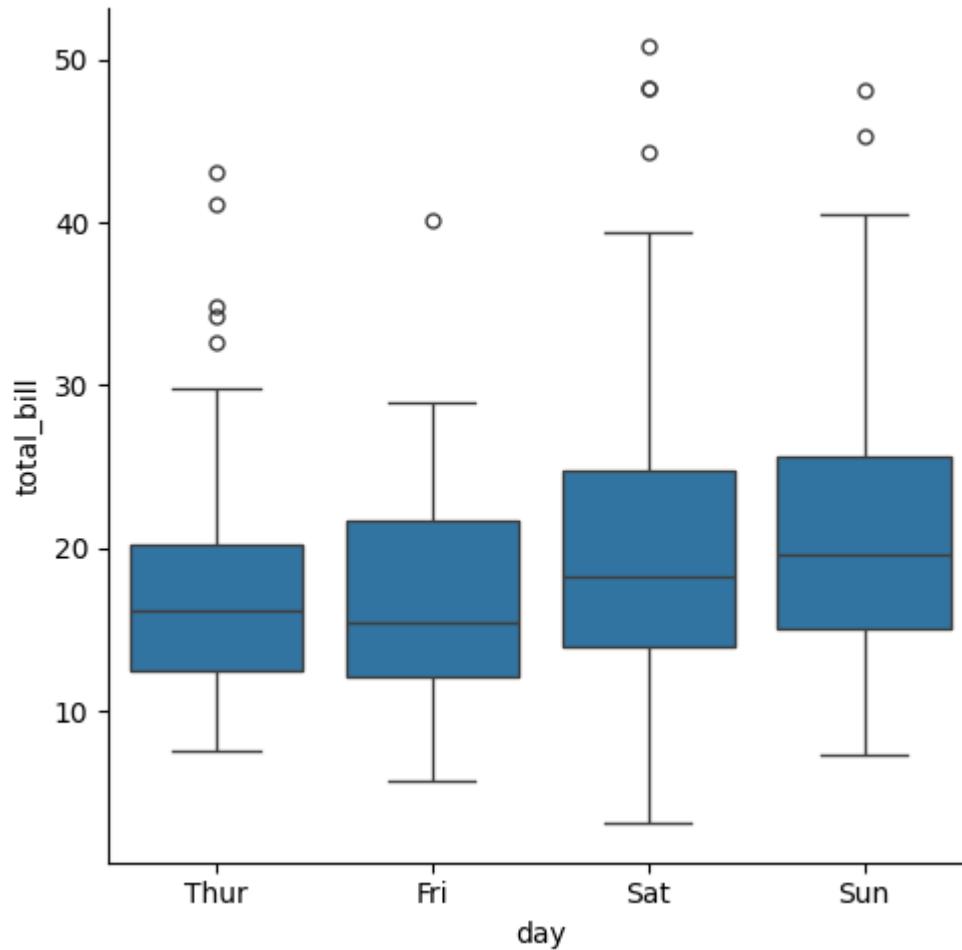
Swarm Plot (swarmplot())

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



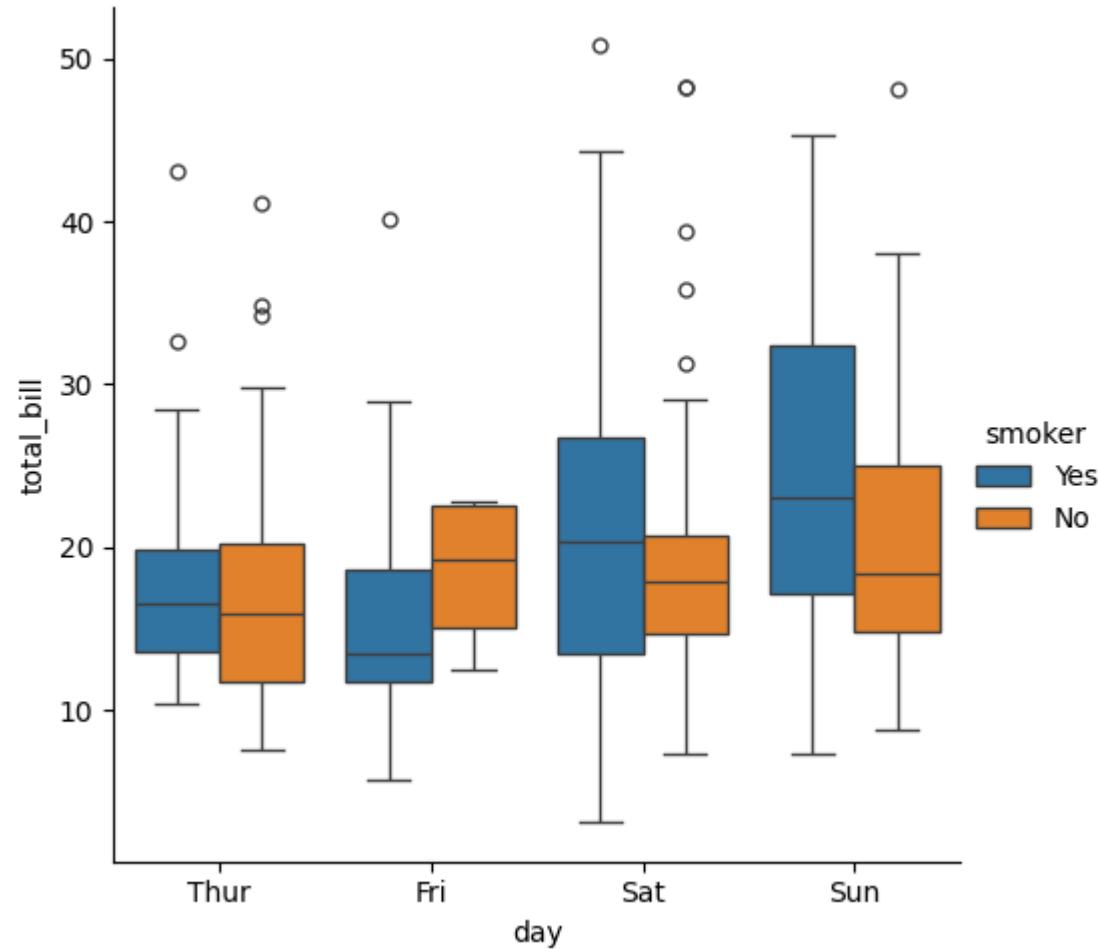
boxplot

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



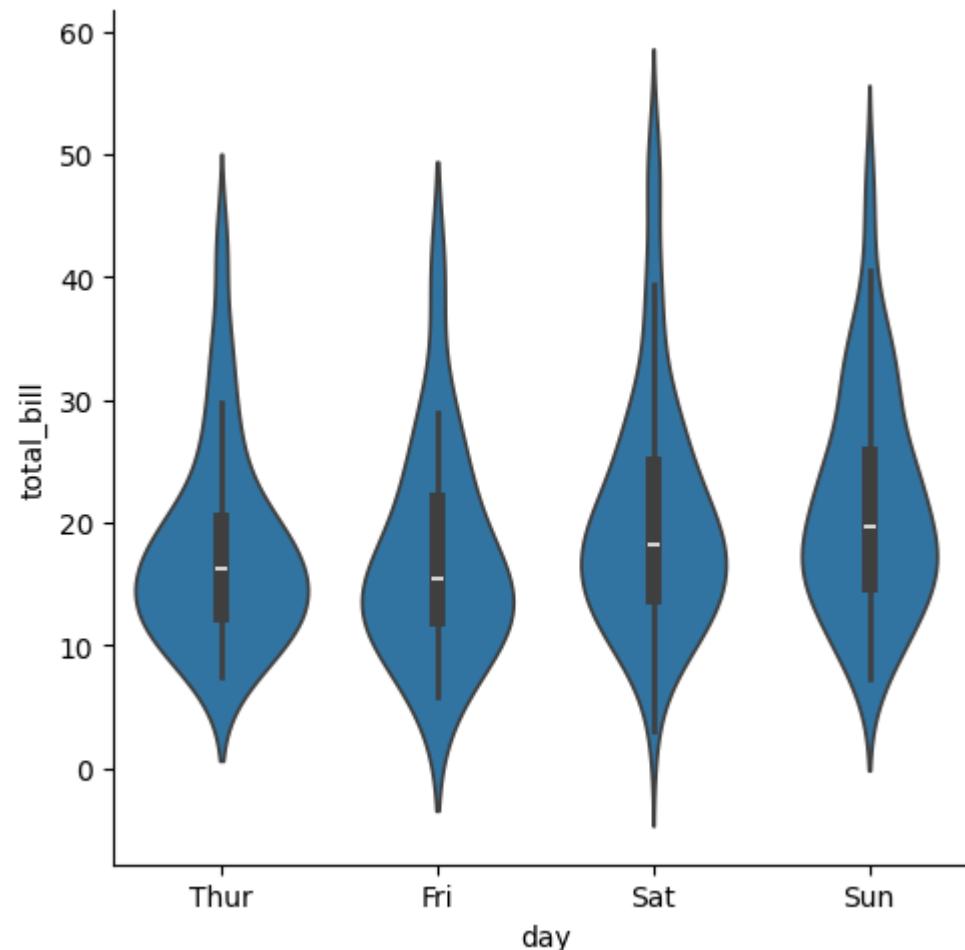
Adjusting hue

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

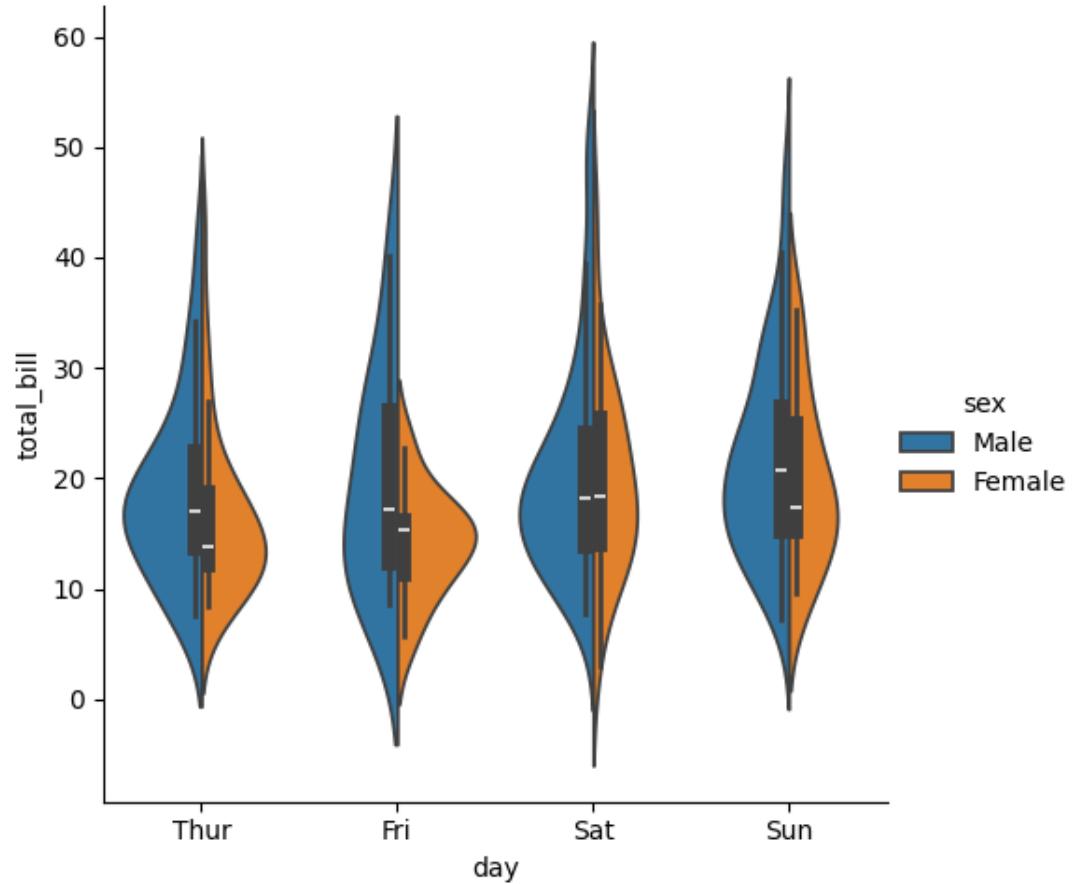


Violin Plot – combines boxplot with kde estimates

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

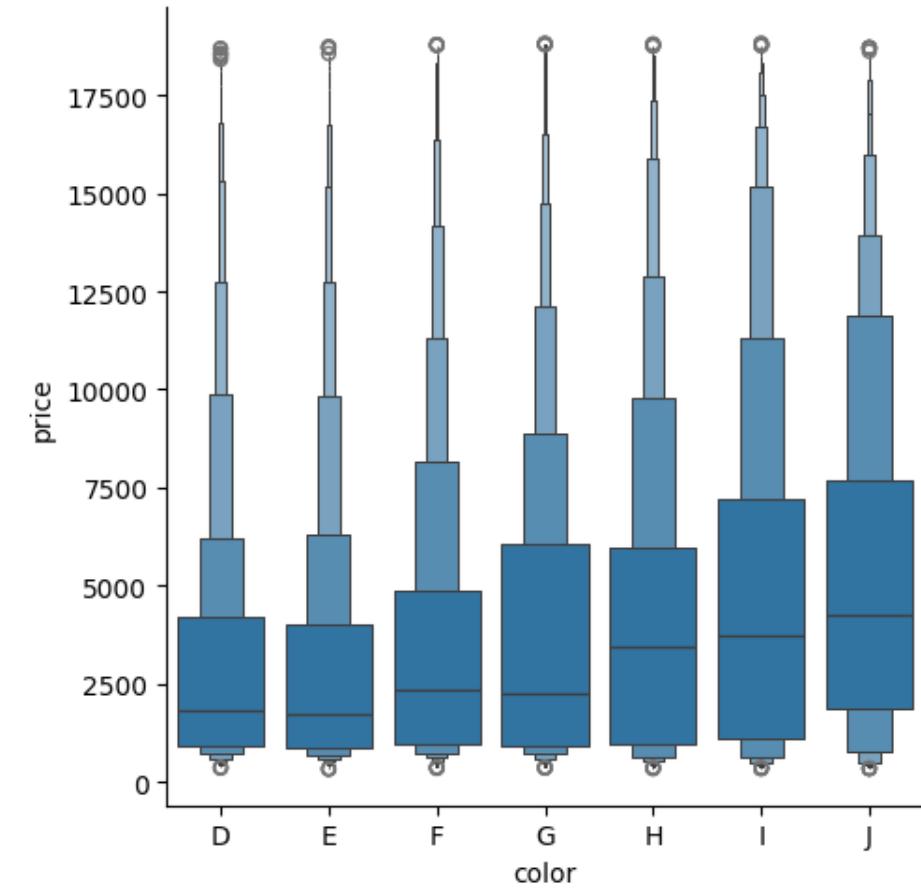


```
sns.catplot(data=tips, x="day", y="total_bill",  
hue="sex", kind="violin", split=True)
```



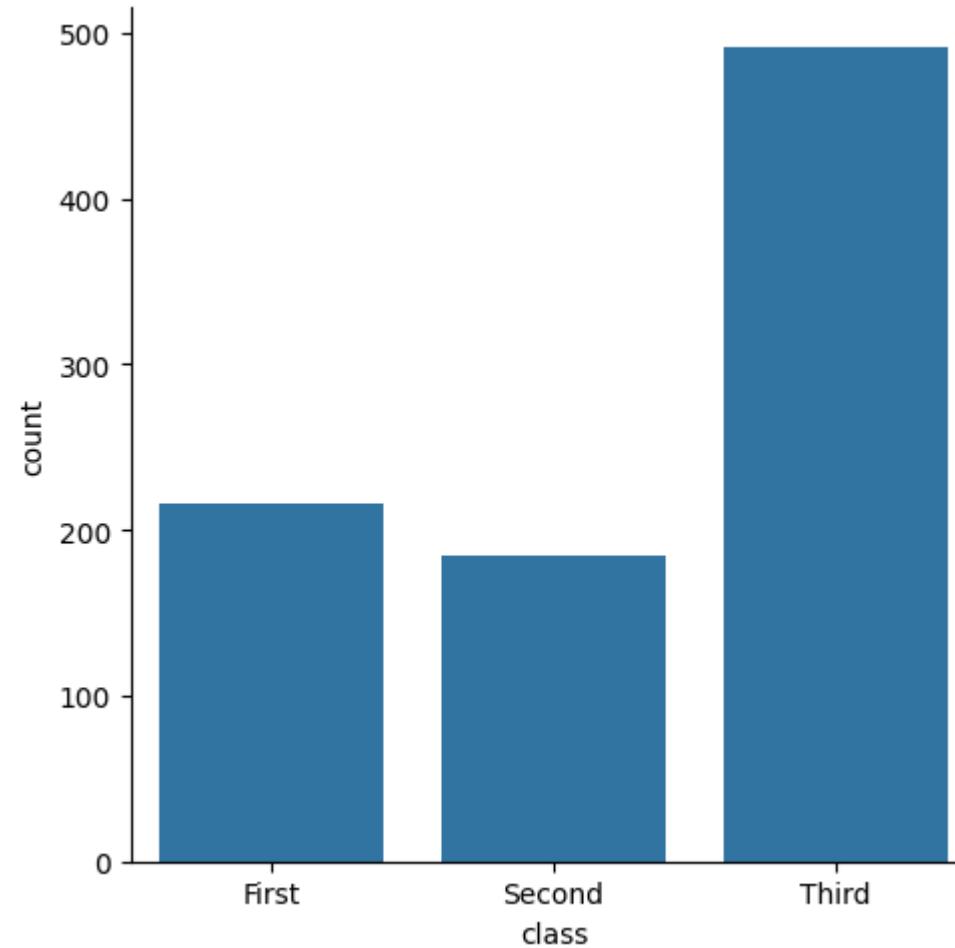
Boxen plot - A box plot alternative for large datasets

```
sns.catplot(data=diamonds, x="color", y="price",  
kind="boxen")
```



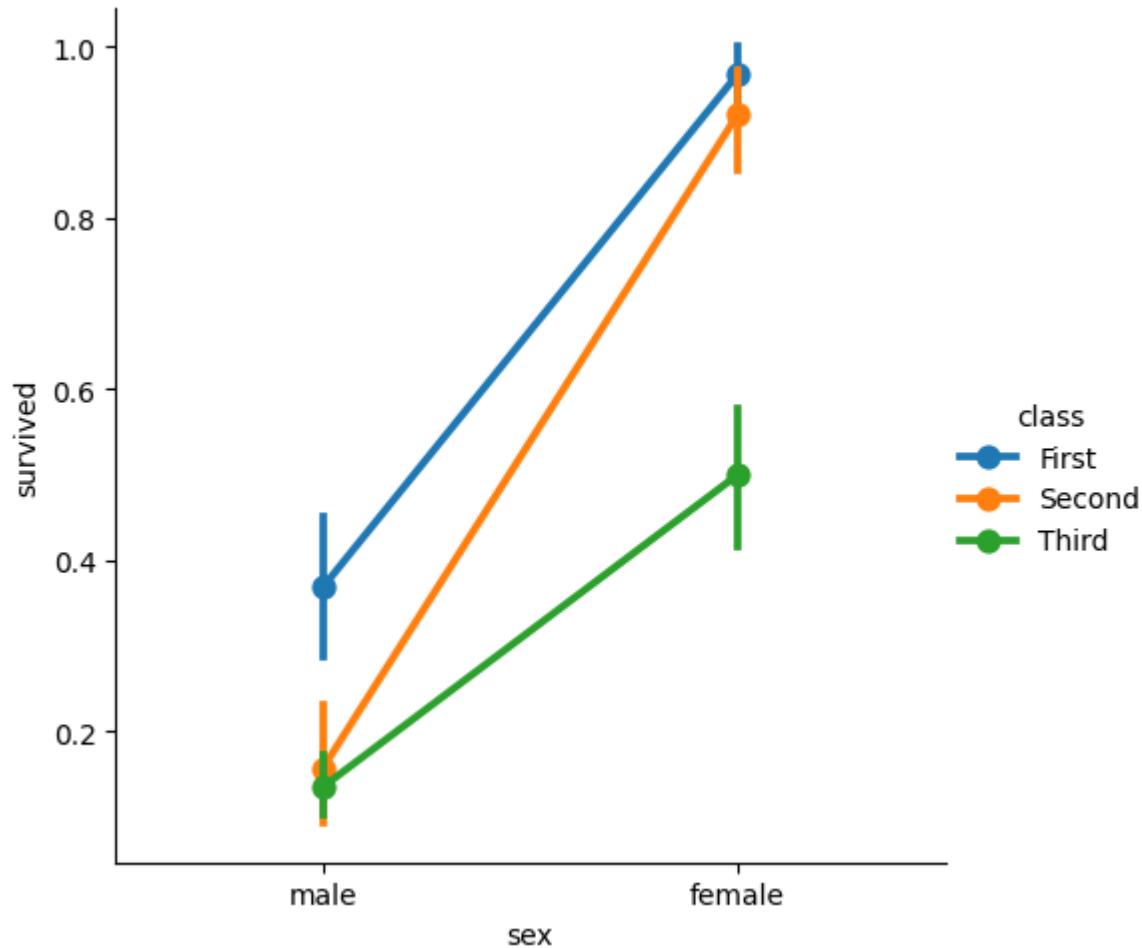
count Plot

```
sns.catplot(data=titanic, x="class", kind="count")
```



Point plot

```
sns.catplot(data=titanic, x="sex", y="survived", hue="class",  
kind="point")
```



Week 7

Time series data

1. Sales Forecasting

- **Example:** A retail company analyzes past sales data to forecast future demand for different products.
- **Method:** ARIMA, Exponential Smoothing, or LSTM models.
- **Use Case:** Inventory optimization, production planning, and budgeting.

2. Stock Market Analysis

- **Example:** A financial analyst examines stock prices and trading volumes to identify trends and make investment decisions.
- **Method:** GARCH models for volatility forecasting.
- **Use Case:** Risk assessment, portfolio management, and trading strategies.

3. Customer Churn Prediction

- **Example:** A telecom company tracks customer subscription data to predict which users are likely to cancel services.
- **Method:** Hidden Markov Models (HMM), Survival Analysis, or Recurrent Neural Networks (RNN).
- **Use Case:** Customer retention strategies and targeted marketing.

4. Website Traffic and User Behavior Analysis

- **Example:** An e-commerce business monitors daily website traffic to optimize marketing campaigns.
- **Method:** Seasonal Decomposition of Time Series (STL), anomaly detection techniques.
- **Use Case:** Digital marketing, ad spend allocation, and conversion rate optimization.

5. Economic Indicators & Business Cycles

- **Example:** A government agency analyzes GDP growth, unemployment rates, and inflation trends.
- **Method:** Vector Autoregression (VAR), Cointegration Analysis.
- **Use Case:** Policy decisions, economic forecasting, and business cycle analysis.

6. Fraud Detection

- **Example:** A bank monitors credit card transactions over time to detect fraudulent activities.
- **Method:** Anomaly detection with Isolation Forests, Bayesian change point detection.
- **Use Case:** Fraud prevention and security enhancement.

7. Demand Planning & Supply Chain Optimization

- **Example:** A logistics company predicts shipping demand fluctuations.
- **Method:** Holt-Winters Exponential Smoothing, Time Series Regression.
- **Use Case:** Efficient resource allocation and delivery scheduling.

What is Time Series Data?

- Time series data is a sequence of observations recorded at **regular time intervals**. Examples include:
- **Stock prices** (daily closing prices)
- **Temperature readings** (hourly weather data)
- **Website traffic** (daily visits)
- **Sales data** (monthly revenue)

Loading and Exploring Time Series Data

- **Using Pandas to Load Time Series Data**

```
import pandas as pd

# Load dataset
df = pd.read_csv("timeseries_data.csv",
parse_dates=["Date"], index_col="Date")

# Display first few rows
print(df.head())
```

- `parse_dates=["Date"]` ensures that the "Date" column is treated as a datetime object.
- `index_col="Date"` sets the Date column as the index.

Working with Dates in Time Series Data

- Converting Strings to Datetime

```
import pandas as pd

date_str = "2024-03-05"
date_obj =
pd.to_datetime(date_str)
print(date_obj)
# Output: 2024-03-05 00:00:00
```

Handling Dates in a DataFrame

- Converting a Column to Datetime Format

```
df = pd.DataFrame({"Date": ["2024-01-01", "2024-02-15", "2024-03-10"]})
```

```
# Convert to datetime  
df["Date"] = pd.to_datetime(df["Date"])
```

```
print(df.dtypes) # Date is now a datetime object
```

Extracting Date Components

- Once converted to datetime format, we can extract datetime components from the data
 - `df["Year"] = df["Date"].dt.year`
 - `df["Month"] = df["Date"].dt.month`
 - `df["Day"] = df["Date"].dt.day`
 - `df["Weekday"] = df["Date"].dt.day_name()`
 - `print(df.head())`

Setting Date as Index

```
df.set_index("Date", inplace=True)  
print(df.head())
```

Filtering data by Date

Filtering by single date

```
df.loc["2024-03-10"]
```

Filtering by range of date values

```
df.loc["2024-01-01":"2024-02-15"]
```

Generating Date Ranges

Daily

```
date_range = pd.date_range(start="2024-01-01", end="2024-12-31", freq="D")
print(date_range)
```

Monthly

```
monthly_range = pd.date_range(start="2024-01-01", periods=12, freq="M")
print(monthly_range)
```

```
DatetimeIndex(['2024-01-31', '2024-02-29',
                 '2024-03-31', '2024-04-30', '2024-05-31',
                 '2024-06-30', '2024-07-31', '2024-08-31',
                 '2024-09-30', '2024-10-31', '2024-11-30',
                 '2024-12-31'],
```

Adding & Subtracting Dates

Adding Days

```
df["Next_Week"] = df.index +  
pd.Timedelta(days=7)
```

Subtracting Dates

```
df["Days_Since"] =  
(pd.Timestamp.today() -  
df.index).days  
(this will calculate no. of days  
since each date)
```

Uses of pd.Timestamp()

- Creating timestamp from string.
- import pandas as pd
- ts = pd.Timestamp("2025-03-24")
- print(ts)
- 2025-03-24 00:00:00

- Creating timestamp from date and time.
- `ts = pd.Timestamp("2025-03-24 14:30:00")`
- `print(ts)`
- `2025-03-24 14:30:00`

Creating timestamp from date components

- ts = pd.Timestamp(year=2025, month=3, day=24, hour=14, minute=30, second=0)
- print(ts)
- 2025-03-24 14:30:00

Getting current timestamp()

- ts_now = pd.Timestamp.now()
- print(ts_now)
- 2025-03-24 14:35:45.567890

Accessing timestamp attributes

- ts = pd.Timestamp("2025-03-24 14:30:00")
- print(ts.year) # 2025
- print(ts.month) # 3
- print(ts.day) # 24
- print(ts.hour) # 14

Performing operations on timestamp

- ts1 = pd.Timestamp("2025-03-24")
- ts2 = ts1 + pd.Timedelta(days=10) # Add 10 days
- print(ts2)
- 2025-04-03 00:00:00

Time Zone Handling

- **Converting Time Zones**

```
df["Date"] = df.index.tz_localize("UTC").tz_convert("US/Eastern")
```

- `df.index.tz_localize("UTC")`If the index does not already have a timezone, this sets it to UTC.
- `.tz_convert("US/Eastern")`Converts the UTC timestamps into the US/Eastern timezone.

Resampling Time Series Data

- Resampling is **changing the frequency** of time series data—either **aggregating** (downsampling) or **spreading out** (upsampling) data points.
- **Examples:**
 - Converting **daily** sales data into **monthly** revenue
 - Converting **minute-level** stock prices into **hourly** trends
 - Filling in **missing timestamps** in incomplete datasets

data

```
import pandas as pd
import numpy as np

# Create sample time series data
date_rng = pd.date_range(start="2024-01-01", periods=100,
freq="D") # Daily data
df = pd.DataFrame({"Date": date_rng, "Sales": np.random.randint(100, 500, size=(100))})

# Convert Date column to index
df.set_index("Date", inplace=True)

print(df.head()) # Check the first few rows
```

Date	Sales
2024-01-01	293
2024-01-02	253
2024-01-03	101
2024-01-04	122
2024-01-05	452

Downsampling

- **Resampling to Monthly Data**

```
df_monthly = df.resample("M").sum() # Sum of Sales per month  
print(df_monthly.head())
```

- **Resampling to Weekly Data**

```
df_weekly = df.resample("W").mean() # Weekly average  
print(df_weekly.head())
```

Upsampling

Resampling to Hourly Data without filling missing values

```
df_hourly = df.resample("H").asfreq() # Keeps missing values as NaN  
print(df_hourly.head())
```

Example before df.resample("H").asfreq()

- Value
- 2025-03-24 08:15:00 10
- 2025-03-24 09:00:00 20
- 2025-03-24 09:45:00 30
- 2025-03-24 10:30:00 40
- 2025-03-24 11:15:00 50

Example after df.resample("H").asfreq()

- Value
- 2025-03-24 08:00:00 NaN
- 2025-03-24 09:00:00 20.0
- 2025-03-24 10:00:00 NaN
- 2025-03-24 11:00:00 50.0

Unsampling

Filling Missing Values After Upsampling

- df_hourly_filled = df.resample("H").ffill() # Forward fill missing values
- print(df_hourly_filled.head())
- **ffill()** Fills missing values with the previous available value.
- **bfill()** → Uses the next available value.
- **Fill with a Default Value (e.g., 0)**
- df_hourly = df.resample("H").asfreq().fillna(0)

- Value
- 2025-03-24 08:00:00 10.0
- 2025-03-24 09:00:00 NaN
- 2025-03-24 10:00:00 20.0
- 2025-03-24 11:00:00 NaN
- 2025-03-24 12:00:00 NaN
- 2025-03-24 13:00:00 NaN
- 2025-03-24 14:00:00 40.0
- 2025-03-24 15:00:00 NaN
- 2025-03-24 16:00:00 50.0

- Value
- 2025-03-24 08:00:00 10.0
- 2025-03-24 09:00:00 10.0 # Filled with 10.0
- 2025-03-24 10:00:00 20.0
- 2025-03-24 11:00:00 20.0 # Filled with 20.0
- 2025-03-24 12:00:00 20.0 # Filled with 20.0
- 2025-03-24 13:00:00 20.0 # Filled with 20.0
- 2025-03-24 14:00:00 40.0
- 2025-03-24 15:00:00 40.0 # Filled with 40.0
- 2025-03-24 16:00:00 50.0

- Value
- 2025-03-24 08:00:00 10.0
- 2025-03-24 09:00:00 20.0 # Filled with next available value
- 2025-03-24 10:00:00 20.0
- 2025-03-24 11:00:00 NaN
- 2025-03-24 12:00:00 40.0 # Filled with next available value
- 2025-03-24 13:00:00 40.0
- 2025-03-24 14:00:00 40.0
- 2025-03-24 15:00:00 50.0 # Filled with next available value
- 2025-03-24 16:00:00 50.0

- Value
- 2025-03-24 08:00:00 10.0
- 2025-03-24 09:00:00 0.0
- 2025-03-24 10:00:00 20.0
- 2025-03-24 11:00:00 0.0
- 2025-03-24 12:00:00 0.0
- 2025-03-24 13:00:00 0.0
- 2025-03-24 14:00:00 40.0
- 2025-03-24 15:00:00 0.0
- 2025-03-24 16:00:00 50.0

Custom Resampling Periods

Frequency Code	Meaning
"D"	Daily
"W"	Weekly
"M"	Monthly
"Q"	Quarterly
"Y"	Yearly
"H"	Hourly
"T"	Every Minute

Rolling window analysis

- **Sales & Revenue Analysis** – Identify sales trends over time
- **Stock Market Analysis** – Moving averages to detect trends
- **Website Traffic Analysis** – Track engagement patterns

Time Series Rolling Window Analysis

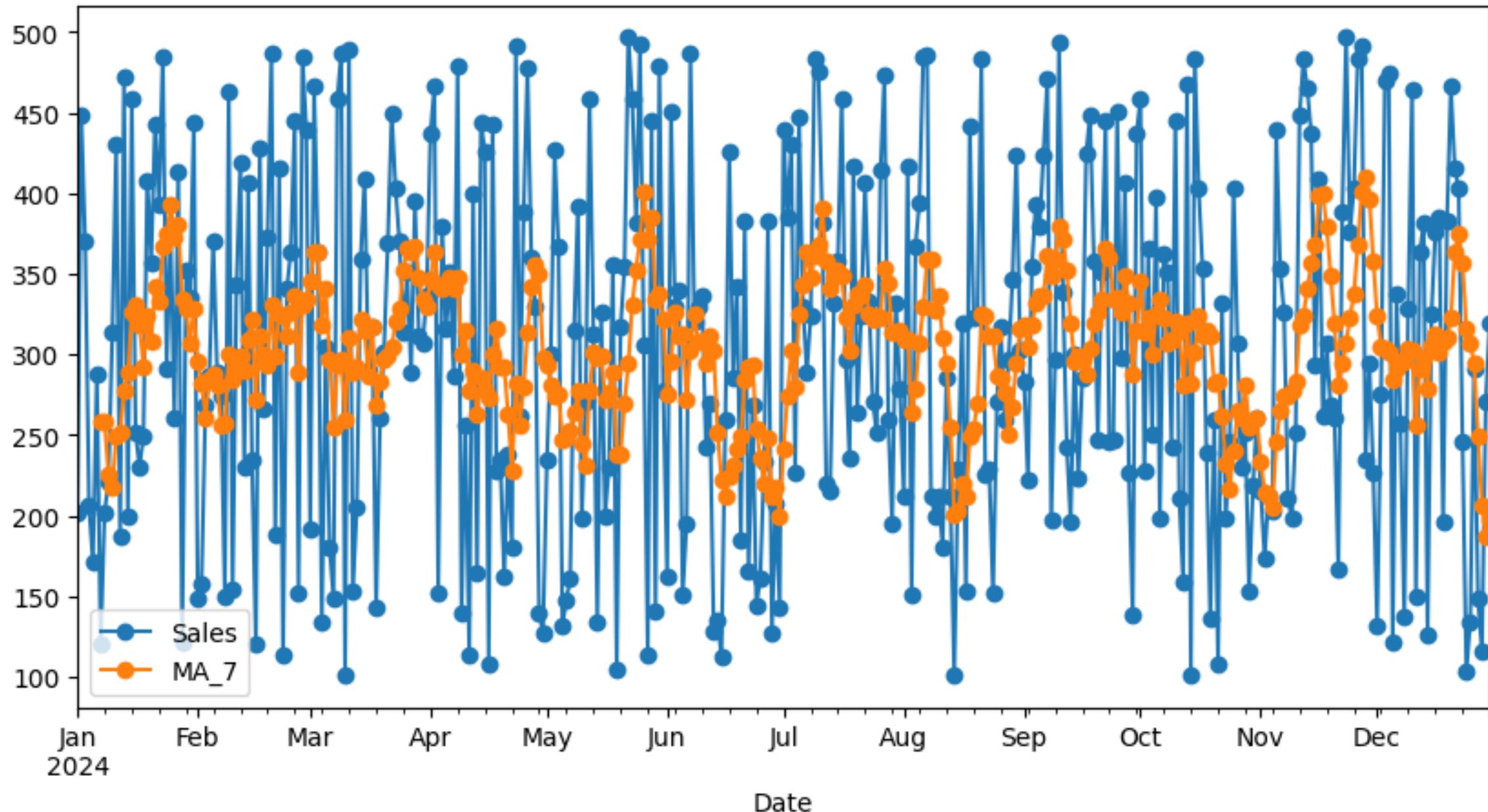
- **Rolling Mean (Moving Average)**

```
df["MA_7"] = df["Sales"].rolling(window=7).mean()
```

```
# Plot sales vs. moving average
df[["Sales", "MA_7"]].plot(figsize=(10, 5), title="7-Day
Moving Average")
plt.show()
```

- **Reduces short-term fluctuations.**
- **Larger window = smoother trend**

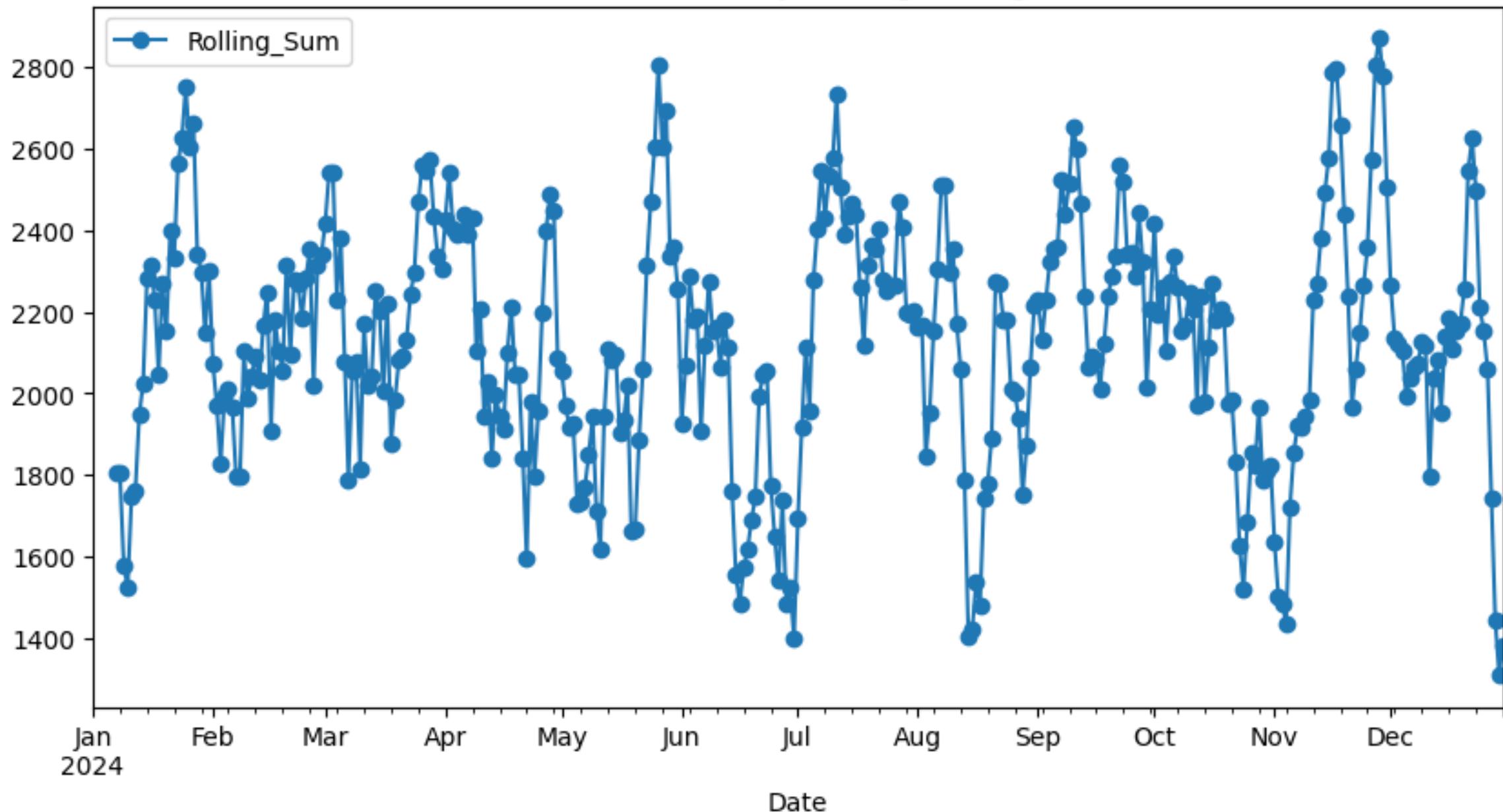
Sales and 7-Day Moving Average



Rolling Sum

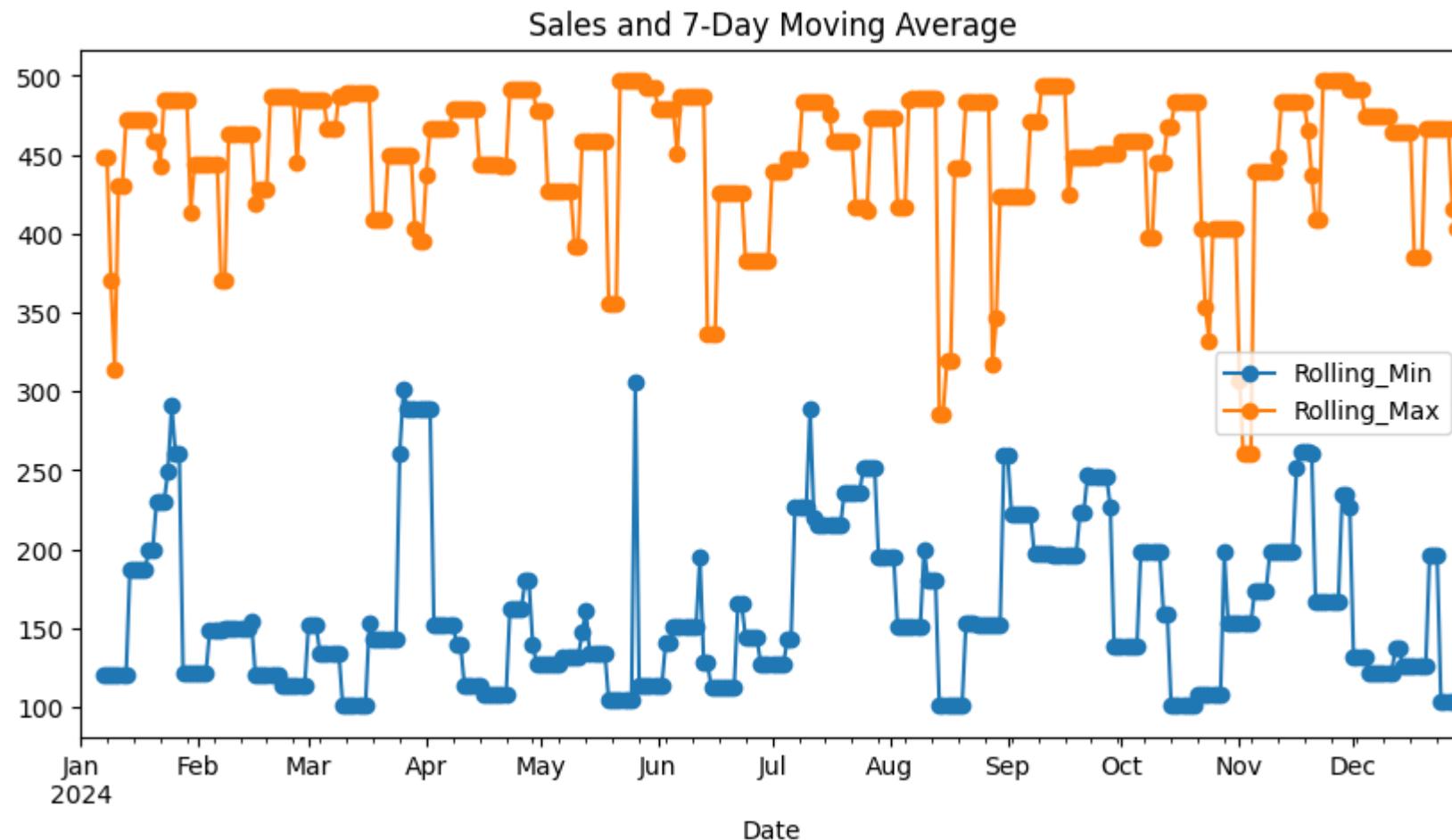
- `df["Rolling_Sum"] = df["Sales"].rolling(window=7).sum()`
- Show cumulative sales over past 7 days

Sales and 7-Day Moving Average



Rolling Min & Max

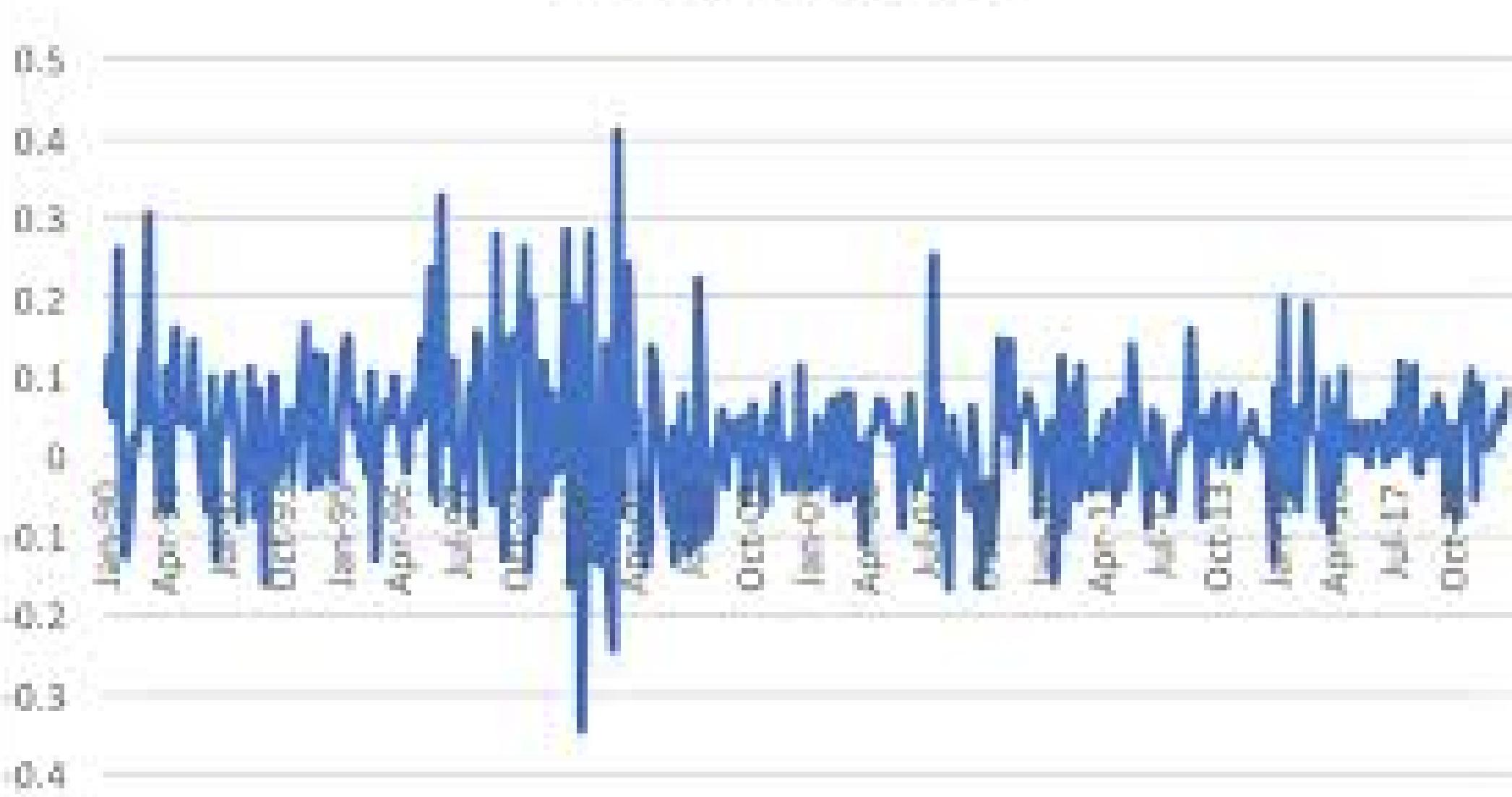
- `df["Rolling_Min"] = df["Sales"].rolling(window=7).min()`
- `df["Rolling_Max"] = df["Sales"].rolling(window=7).max()`
- **Useful for identifying the lowest & highest sales in a rolling period.**

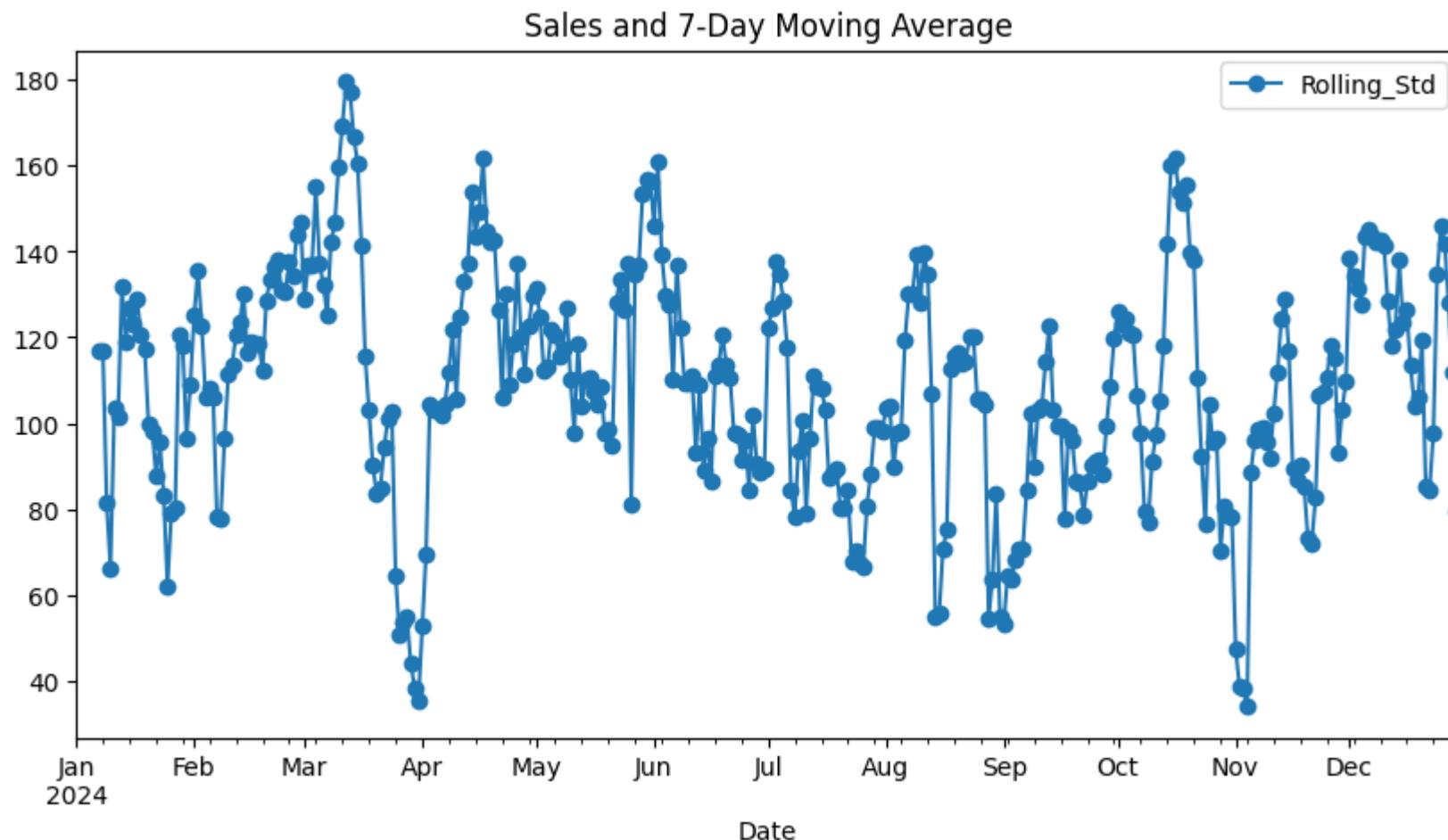


Weighted Moving Average (More Weight to Recent Data)

- `df["WMA_7"] = df["Sales"].ewm(span=7, adjust=False).mean()`
 - **Exponential Weighted Moving Average (EWMA)** gives **more weight to recent values**
 - **EWMA is useful in volatile data** (e.g., stock prices, sales trends) to react **faster to market changes**.
-
- **Rolling Standard Deviation**
 - `df["Rolling_Std"] = df["Sales"].rolling(window=7).std()`
 - **Shows variability in sales over time.**

Microsoft returns

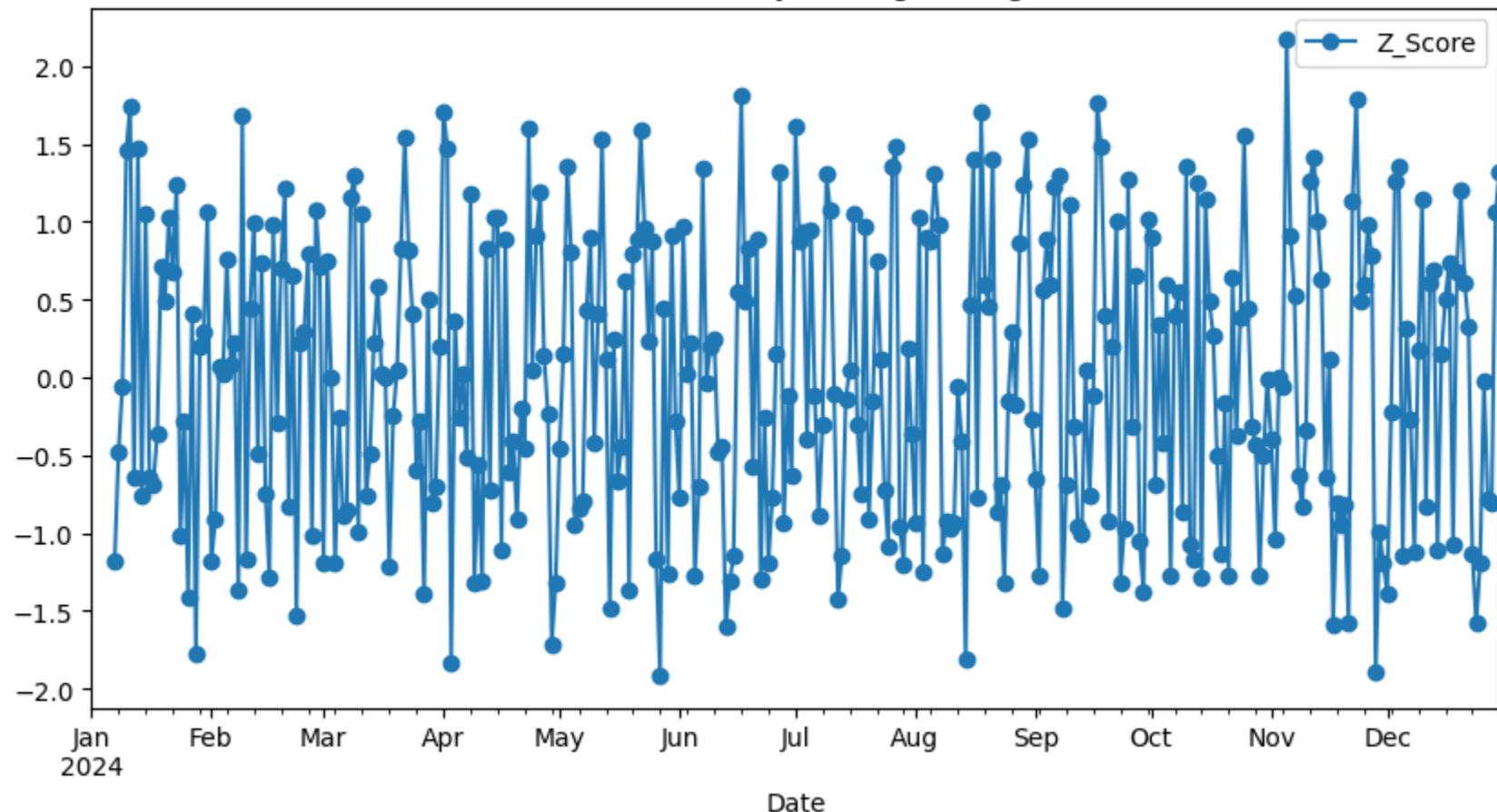




Detecting Anomalies with Rolling Z-Scores

- `df["Rolling_Mean"] = df["Sales"].rolling(window=7).mean()`
- `df["Rolling_Std"] = df["Sales"].rolling(window=7).std()`
- `df["Z_Score"] = (df["Sales"] - df["Rolling_Mean"]) / df["Rolling_Std"]`
- **# Flag anomalies where Z-Score > 2**
- `df["Anomaly"] = df["Z_Score"].apply(lambda x: "Yes" if abs(x) > 2 else "No")`
- `print(df[df["Anomaly"] == "Yes"])`
- **Identifies points that are significantly different from the rolling mean.**

Sales and 7-Day Moving Average



Week 8

Data Merging and Joining

Difference between merging and joining

- Both merging and joining combine data from multiple DataFrames, but they differ in how they're implemented and when they're most convenient:
- **Merging (using pd.merge()):**
 - **Key-based:** You merge DataFrames based on one or more common columns (keys).
 - **Flexible Join Types:** Supports inner, left, right, and outer joins.
 - **Customizable:** You can specify which columns to merge on, even if they have different names in the two DataFrames (using `left_on` and `right_on`).

Difference between merging and joining

- **Joining (using DataFrame.join()):**
 - **Index-based:** It is primarily used to join DataFrames on their indexes.
 - **Simpler Syntax:** It's a convenient shorthand for index-based merges.
 - **Can Use Columns:** You can join on columns, but you need to set one of the DataFrame's index first.

Different methods

- **pd.merge()** lets you combine two DataFrames based on one or more common columns (similar to SQL joins).
- **DataFrame.join()** is a convenient method for joining two DataFrames based on their indexes.
- **pd.concat()** is used to stack DataFrames either vertically (adding rows) or horizontally (adding columns). It's not based on keys but rather on positions.

pd.merge()

- pd.merge() is a powerful function in Pandas that lets you combine two DataFrames based on one or more common keys, similar to SQL joins. It gives you a lot of flexibility in specifying how to join your data. Here's a detailed overview:

- **Key Features of pd.merge()**
- **Key-Based Merging:**
 - Merge DataFrames using one or more columns as keys.
 - These keys can be of the same name in both DataFrames or you can specify different key names using `left_on` and `right_on`.

Join Types:

- You can perform:
- **Inner Join** (default): Returns rows where the keys exist in both DataFrames.
- **Left Join**: Returns all rows from the left DataFrame, along with matching rows from the right DataFrame. Missing values are filled with NaN.
- **Right Join**: Returns all rows from the right DataFrame, along with matching rows from the left DataFrame.
- **Outer Join**: Returns all rows from both DataFrames, filling in NaNs where there are no matches.

- **Handling of Suffixes:**
- When merging, if there are overlapping column names not used as keys, you can specify suffixes using the suffixes parameter.
- **Multiple Keys:** You can merge on multiple columns by passing a list to the on parameter.

Basic syntax

- `pd.merge(left, right, how='inner', on=None, left_on=None, right_on=None, left_index=False, right_index=False, sort=False, suffixes=('_x', '_y'))`.
- **Left** and **right** are the two datasets.
- **How** specifies join types:
- **Inner, outer, left and right.**
- **Inner join:**
- `merged_inner = pd.merge(df1, df2, on='ID', how='inner')`

Left Join with Different Key Names

- df1 = pd.DataFrame({
 - 'EmpID': [1, 2, 3],
 - 'Name': ['Alice', 'Bob', 'Charlie']})
- df2 = pd.DataFrame({
 - 'ID': [2, 3, 4],
 - 'Score': [88, 92, 75]})
- merged_left = pd.merge(df1, df2, left_on='EmpID', right_on='ID', how='left')

Merge on Multiple Keys

- df1 = pd.DataFrame({
 - 'ID': [1, 2, 3, 4],
 - 'Dept': ['HR', 'IT', 'IT', 'HR'],
 - 'Name': ['Alice', 'Bob', 'Charlie', 'David']]})
 - df2 = pd.DataFrame({
 - 'ID': [2, 3, 4, 4],
 - 'Dept': ['IT', 'IT', 'HR', 'HR'],
 - 'Salary': [60000, 65000, 70000, 72000]]})
 - merged_multi = pd.merge(df1, df2, **on=['ID', 'Dept']**, how='inner')

- Merge on Multiple Keys:

	ID	Dept	Name	Salary
•	0 2	IT	Bob	60000
•	1 3	IT	Charlie	65000
•	2 4	HR	David	70000

Merging on index

- df1 = pd.DataFrame({
 - 'Name': ['Alice', 'Bob', 'Charlie']
- }, index=[1, 2, 3])
- df2 = pd.DataFrame({
 - 'Score': [88, 92, 75]
- }, index=[2, 3, 4])
- merged_index = pd.merge(df1, df2, left_index=True, right_index=True, how='inner')

Output- index merging

- Merge on Index:
 - Name Score
 - 2 Bob 88
 - 3 Charlie 92

DataFrame.join()

- **DataFrame.join()** is a convenient method in Pandas for combining DataFrames primarily by their indexes.
- It is particularly useful when you want to merge data along the row labels without needing to specify explicit key columns.

Key Features of DataFrame.join()

- **Index-Based Joining:**
- df.join() merges two DataFrames using their indexes. If the DataFrames have different indexes, you can use different join types (like 'inner' or 'outer').
- **Combining Multiple DataFrames:**
 - You can join more than two DataFrames by passing a list of DataFrames.
- **Column Alignment:**
 - When the DataFrames share some column names (other than the index), you can control suffixes to distinguish them.
- **Convenience:**
 - join() is often a shorthand for merging on the index when you already have your data set up with the appropriate indexes.

Basic Syntax

- DataFrame.join(other, on=None, how='left', lsuffix='', rsuffix='', sort=False)
- **other:** DataFrame or list/tuple of DataFrames to join.
- **on:** Column or index level name to.
- **how:** Type of join:
 - 'left' (default): Keep all rows from the calling DataFrame.
 - 'right': Keep all rows from the other DataFrame.
 - 'outer': Keep all rows from both DataFrames.
 - 'inner': Keep only rows that have matching indexes in both DataFrames.
- **lsuffix, rsuffix:** Suffixes to apply to overlapping column names.sort:
Sort the resulting DataFrame by the join keys.

Simple Index-Based Join

- import pandas as pd
- df1 = pd.DataFrame({
- 'Name': ['Alice', 'Bob', 'Charlie']
- }, index=[1, 2, 3])
- df2 = pd.DataFrame({
- 'Score': [88, 92, 75]
- }, index=[2, 3, 4])
- joined_df = df1.join(df2, how='left')

Output

- Left Join (using df1's index):

	Name	Score
• 1	Alice	NaN
• 2	Bob	88.0
• 3	Charlie	92.0

Inner Join on Index

- joined_inner = df1.join(df2, how='inner')
- Inner Join (common index only):
 - Name Score
 - 2 Bob 88.0
 - 3 Charlie 92.0

Joining Multiple DataFrames

- df3 = pd.DataFrame({
 - 'Department': ['HR', 'IT', 'Finance']
- }, index=[1, 2, 3])
- **# Join multiple DataFrames using the join() method**
- combined_df = df1.join([df2, df3], how='outer')
- print("Join multiple DataFrames:\n", combined_df)

Output

Join multiple DataFrames:

	Name	Score	Department
1	Alice	NaN	HR
2	Bob	88.0	IT
3	Charlie	92.0	Finance
4	NaN	75.0	NaN

Using a Column for Joining

- If you want to join on a column instead of the index, set the index of one DataFrame accordingly. For example:
- # Reset index so that 'ID' becomes a column
- df1_reset = df1.reset_index().rename(columns={'index': 'ID'})
- df2_reset = df2.reset_index().rename(columns={'index': 'ID'})
- # Set 'ID' as index for joining
- df1_reset.set_index('ID', inplace=True)
- df2_reset.set_index('ID', inplace=True)
- joined_on_column = df1_reset.join(df2_reset, how='outer')
- print("Join using a column (set as index):\n", joined_on_column)

Pd.concat()

- pd.concat() is a Pandas function used to combine DataFrames (or Series) along a particular axis. It's very flexible and useful when you want to stack data either vertically (i.e., adding rows) or horizontally (i.e., adding columns).

Key Features

- **Axis Control:**
 - Vertical Concatenation (axis=0): Stacks DataFrames on top of each other (like appending rows).
 - Horizontal Concatenation (axis=1): Combines DataFrames side-by-side (like adding columns).
- **Ignore Index:** By default, the original row labels are retained. Setting `ignore_index = True` will reset the index in the result.
- **Keys and Hierarchical Indexing:** You can assign keys to each DataFrame being concatenated, which will create a hierarchical (MultiIndex) on the resulting axis.
- **Joining:** When concatenating along columns, you can control how indexes are aligned using the `join` parameter ('inner' or 'outer').

pd.concat(objs, axis=0, join='outer', ignore_index=False, keys=None)

- **objs:**A list (or dictionary) of DataFrames or Series to concatenate.
- **axis:**The axis to concatenate along. axis=0 stacks rows; axis=1 stacks columns.
- **join:**How to handle indexes on the other axis: 'outer' (union) or 'inner' (intersection).
- **ignore_index:**If True, do not use the existing index values. The result will have a default integer index.
- **keys:**Used to create a hierarchical index on the concatenation axis. Useful for identifying which DataFrame each row/column came from.

Vertical concatenation (Default)

- import pandas as pd
- df1 = pd.DataFrame({'A': ['A0', 'A1'], 'B': ['B0', 'B1']})
- df2 = pd.DataFrame({'A': ['A2', 'A3'], 'B': ['B2', 'B3']})
- result = pd.concat([df1, df2])
- print(result)

Output

A B

- 0 A0 B0
- 1 A1 B1
- 0 A2 B2
- 1 A3 B3

Vertical Concatanation with ignore_index

- result = pd.concat([df1, df2], ignore_index=True)
- print(result)
 - A B
- 0 A0 B0
- 1 A1 B1
- 2 A2 B2
- 3 A3 B3

Horizontal concatenation

- df3 = pd.DataFrame({'C': ['C0', 'C1']})
- result = pd.concat([df1, df3], axis=1)
- print(result)

	A	B	C
0	A0	B0	C0
1	A1	B1	C1

Inner Join (only common columns or indexes)

- df4 = pd.DataFrame({'B': ['B2', 'B3'], 'C': ['C2', 'C3']})
- result = pd.concat([df1, df4], join='inner')
- print(result)
 - B
 - 0 B0
 - 1 B1
 - 0 B2
 - 1 B3

Using Keys (for multi-index)

- `result = pd.concat([df1, df2], keys=['df1', 'df2'])`
- `print(result)`

	A	B
• df1 0	A0	B0
• 1	A1	B1
• df2 0	A2	B2
• 1	A3	B3

SQL Basics & Joins - Detailed Lecture

Week 9

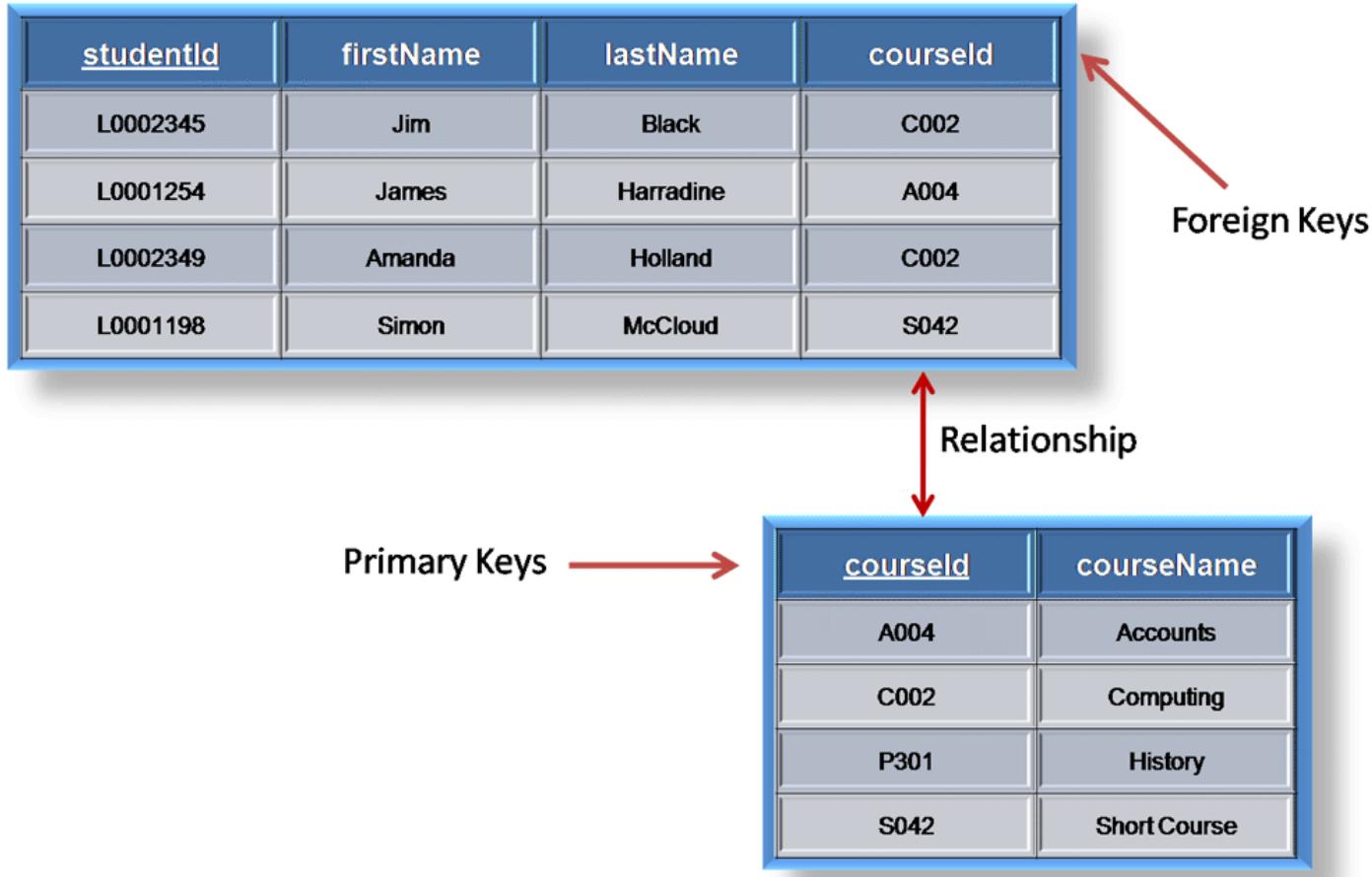
Database

- A **database** is an organized collection of data that allows users to store, retrieve, and manage information efficiently. Databases can be structured (like relational databases) or unstructured (like NoSQL databases). They serve various purposes, from tracking business transactions to storing user data for applications.
- A **relational database** is a type of structured database that organizes data into tables (relations) with predefined columns and rows. It uses **relationships** between tables to efficiently link and retrieve data. Relational databases follow the principles of **Structured Query Language (SQL)** for querying and manipulating data.

Relational databases

- Key characteristics of relational databases:
- **Tables:** Store data in structured rows and columns.
- **Keys:** Primary keys uniquely identify each row, while foreign keys establish relationships between tables.
- **Normalization:** Reduces redundancy and ensures data integrity by structuring information across multiple tables.
- **SQL Usage:** Queries like SELECT, JOIN, GROUP BY, and ORDER BY help retrieve meaningful insights.

Primary



Normalization

The First Normal Form – 1NF

For a table to be in the first normal form, it must meet the following criteria:

- a single cell must not hold more than one value (atomicity)
- there must be a primary key for identification
- no duplicated rows or columns
- each column must have only one value for each row in the table

The Second Normal Form – 2NF

The 1NF only eliminates repeating groups, not redundancy. That's why there is 2NF.

A table is said to be in 2NF if it meets the following criteria:

- it's already in 1NF
- has no partial dependency. That is, all non-key attributes are fully dependent on a primary key.

The Third Normal Form – 3NF

When a table is in 2NF, it eliminates repeating groups and redundancy, but it does not eliminate transitive partial dependency.

This means a non-prime attribute (an attribute that is not part of the candidate's key) is dependent on another non-prime attribute. This is what the third normal form (3NF) eliminates.

So, for a table to be in 3NF, it must:

- be in 2NF
- have no transitive partial dependency.

Student ID	Name	Subjects
1	Alice	Physics, Math
2	Bob	Chemistry, Biology

Student ID	Name	Subject
1	Alice	Physics
1	Alice	Math
2	Bob	Chemistry
2	Bob	Biology

2NF

Student_ID	Course_ID	Student_Name	Course_Name
1	CS101	Alice	Algorithms
1	CS102	Alice	Databases
2	CS101	Bob	Algorithms

Student_ID	Student_Name	Course_ID	Course_Name
1	Alice	CS101	Algorithms
2	Bob	CS102	Databases

Student_ID	Course_ID
1	CS101
1	CS102
2	CS101

Student_ID	Student_Name	Department_ID	Department_Name
1	Alice	D01	Computer Science
2	Bob	D02	Mathematics

Student_ID	Student_Name	Department_ID
1	Alice	D01
2	Bob	D02

Department_ID	Department_Name
D01	Computer Science
D02	Mathematics

What is SQL?

- SQL stands for Structured Query Language.
- It is used for storing, manipulating, and retrieving data in relational databases.
- SQL is a standard language supported by most RDBMS such as MySQL, SQL Server, and SQLite.

Applications of SQL

- Data analysis and reporting
- Data warehousing
- Backend for web and mobile apps
- Data integration in Big Data systems

RDBMS Overview

- Relational Database Management Systems store data in tables.
- Each table consists of rows and columns.
- Tables can be linked using relationships.

Basic SQL Syntax

- SQL statements are case-insensitive but typically written in uppercase.
- Statements end with a semicolon ';'.
• Keywords: **SELECT, FROM, WHERE, GROUP BY, ORDER BY, JOIN**, etc.

SELECT Statement - Basic

- **SELECT** is used to fetch data from a database.
- Syntax: **SELECT column1, column2 FROM table_name;**
- Use '*' to select all columns: **SELECT * FROM table_name;**

SELECT Statement - Practice

- Given a table 'students':
- **SELECT name, age FROM students;**
- **SELECT * FROM students WHERE age > 18;**

WHERE Clause - Basics

- Used to filter records based on specific conditions.
- Operators: `=`, `<>`, `>`, `<`, `>=`, `<=`, **BETWEEN**, **LIKE**, **IN**
- Example: **SELECT * FROM employees WHERE department = 'Sales';**

WHERE Clause - Logical Operators

- **AND:** Combines conditions that must both be true.
- **OR:** Either condition can be true.
- **NOT:** Negates a condition.
- Example: **SELECT * FROM employees WHERE age > 30 AND department = 'HR';**

WHERE with BETWEEN, LIKE, IN

- **BETWEEN:** `SELECT * FROM employees WHERE age BETWEEN 25 AND 35;`
- **LIKE:** `SELECT * FROM employees WHERE name LIKE 'A%';`
- % means any number of characters (including zero).

id	Name	Department	Salary
1	Alice	HR	60000
2	Andrew	IT	75000
3	Amanda	Sales	52000

IN

- **SELECT * FROM** employees
WHERE department **IN**
('HR', 'Sales');
- **IN** is a cleaner alternative
to multiple **OR** conditions,
Equivalent to:
- department = 'HR' **OR**
department = 'Sales'

id	name	department	salary
1	Alice	HR	60000
3	Amanda	Sales	52000
5	Brian	HR	58000

GROUP BY - Basics

- **GROUP BY** groups rows that have the same values in specified columns.
- Often used with aggregation functions:
COUNT, SUM, AVG, MAX, MIN
- Example: **SELECT department, COUNT(*) FROM employees GROUP BY department;**

GROUP BY with COUNT Example

- **SELECT department, COUNT(*) FROM employees GROUP BY department;**
- This query counts how many employees are in each department.
- Example Output:
- HR → 3 employees
- Sales → 5 employees
- IT → 4 employees
- Useful for reports and summaries where aggregated data is needed.

GROUP BY - Practice

- **SELECT age, COUNT(*) FROM students GROUP BY age;**
- **SELECT department, AVG(salary) FROM employees GROUP BY department;**

ORDER BY - Basics

- **ORDER BY** sorts the results returned by the query.
- Default order is ascending (ASC), use DESC for descending.
- Example: **SELECT name, age FROM employees ORDER BY age DESC;**

ORDER BY - Multiple Columns

- **ORDER BY** can sort by multiple columns.
- Example: **SELECT name, department **FROM** employees **ORDER BY** department, name **ASC**;**

Combining **SELECT**, **WHERE**, **GROUP BY**, **ORDER BY**

- Example:
- **SELECT department, COUNT(*)**
FROM employees WHERE salary >
50000 GROUP BY department
ORDER BY COUNT(*) DESC;

Common Errors and Tips

- Check column names and table names for typos.
- Use semicolons to terminate statements.
- Test queries step-by-step.

Introduction to Joins

- Joins are used to retrieve data from multiple related tables.
- Types: INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN
- Each join type defines how unmatched rows are handled.

INNER JOIN - Basics

- Returns rows with matching values in both tables.
- Syntax: **SELECT** columns **FROM** table1 **INNER JOIN** table2 **ON** table1.column = table2.column;

INNER JOIN - Example

- **SELECT employees.name,
departments.name FROM employees
INNER JOIN departments ON
employees.dept_id = departments.id;**

LEFT JOIN - Basics

- Returns all records from the left table and matched records from the right table.
- Unmatched rows from the right table return NULL.

LEFT JOIN - Example

- **SELECT** employees.name,
departments.name **FROM** employees
LEFT JOIN departments **ON**
employees.dept_id = departments.id;

RIGHT JOIN - Basics

- Returns all records from the right table and matched records from the left table.
- Unmatched rows from the left table return NULL.

FULL JOIN - Basics

- Combines results of both LEFT and RIGHT JOIN.
- Returns all records when there is a match in either left or right table.

FULL JOIN - Example

- `SELECT e.name, d.name FROM employees e FULL JOIN departments d ON e.dept_id = d.id;`

CROSS JOIN - Basics

- Returns Cartesian product of two tables.
- Every row of the first table is joined to every row of the second table.

CROSS JOIN - Example

- `SELECT * FROM employees CROSS JOIN departments;`

Using Aliases in Joins

- Shorten query and improve readability.
- Example: `SELECT e.name, d.name
FROM employees AS e JOIN
departments AS d ON e.dept_id =
d.id;`

Nesting Joins

- Multiple joins can be chained.
- Example: **SELECT e.name, d.name, l.city** **FROM** employees e **JOIN** departments d **ON** e.dept_id = d.id **JOIN** locations l **ON** d.loc_id = l.id;

Join Practice - 1

- Use INNER JOIN to fetch employee names with their department names.

Join Practice - 1

- Use **INNER JOIN** to fetch employee names with their department names.
- **SELECT e.name AS employee_name, d.name AS department_name FROM employees e INNER JOIN departments d ON e.dept_id = d.id;**

Join Practice - 2

- Use LEFT JOIN to list all employees and their departments, even if the department is missing.

Join Practice - 3

- Combine **JOIN** with **WHERE** and **ORDER BY**.
- Example: `SELECT e.name, d.name
FROM employees e JOIN
departments d ON e.dept_id = d.id
WHERE d.name = 'Sales' ORDER BY
e.name;`

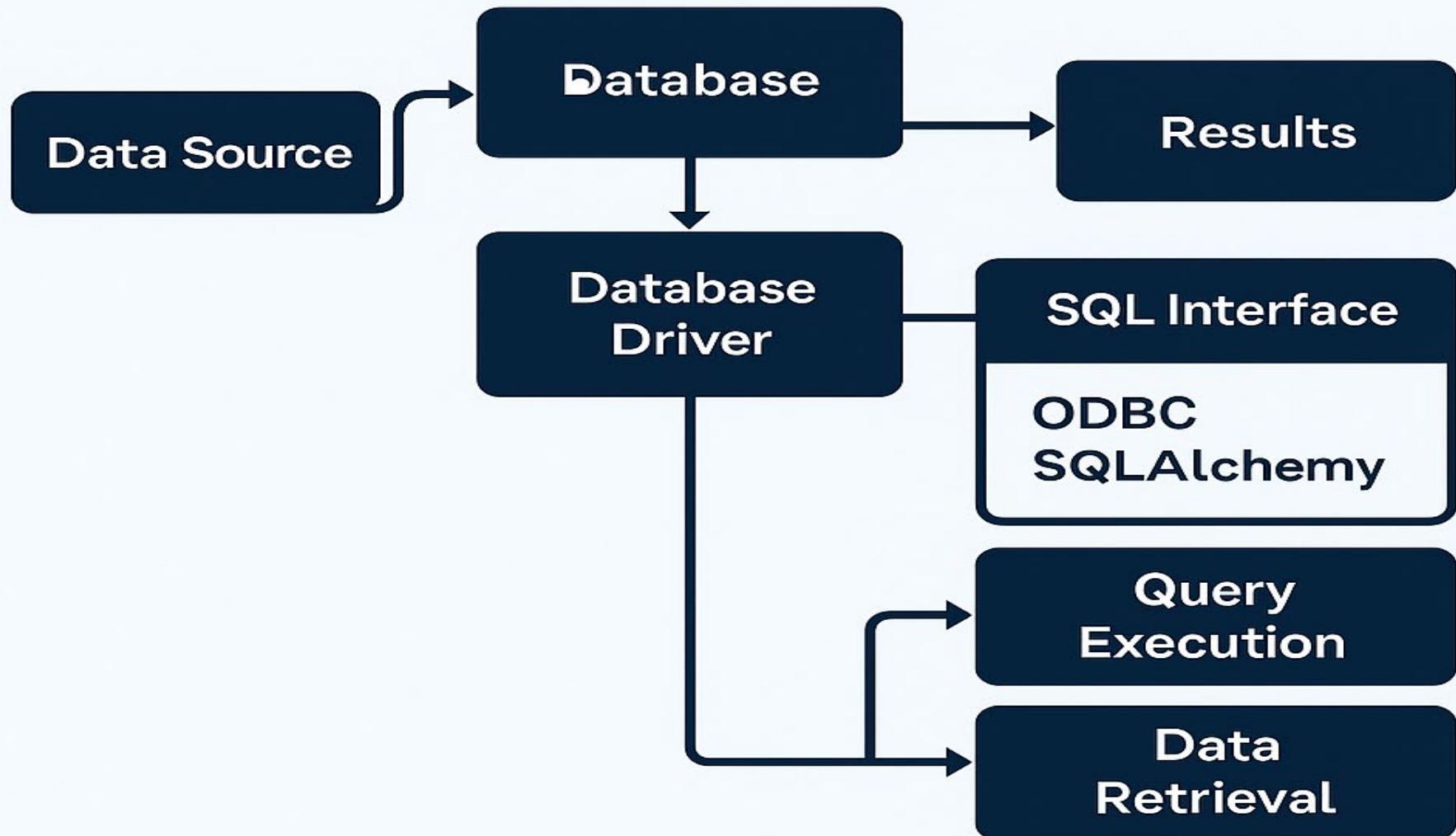
Join class Questions

- 1. What's the result of an INNER JOIN with no matching keys?
- 2. How does a LEFT JOIN differ from a FULL JOIN?
- 3. Write a query that shows departments without employees.

Week 10: Using Python and SQL Together

Business Data Analysis - MGT-174

Using Python and SQL Together



Introduction to Python and SQL Integration

- **SQL** excels at structured data storage, fast querying, and filtering.
- **Python** is ideal for data analysis, statistical modeling, and machine learning.
- **Together**, they enable end-to-end workflows: data extraction → transformation → analysis → visualization.

Use Cases in Business Analytics

- Reporting and visualization
- Data cleaning and preprocessing (scaling etc.)
- Real-time analytics - scheduled queries to retrieve live data and analyze it using python

Data Engineering vs Data Analysis

- **ETL vs Data science workflows**
 - ETL ensures the **right data** gets to the right place in the right format.
Data science uses that data to **generate insights and predictions**.
- **SQL:** Query, join, filter data before loading
- **Python:** uses advanced Machine learning, statistics, data viz, model

Database Connectors Overview

Database connectors are libraries or drivers that allow Python to communicate with SQL databases. They handle the connection, authentication, and data transmission between your Python code and the database engine.

Connector Type	Description	Example Libraries
Built-in	Comes with Python or minimal setup	sqlite3
Native Drivers	Database-specific connectors	psycopg2, pymysql, mysql-connector-python
Big Data/Cloud	For cloud-based and distributed systems	snowflake-connector-python, bigquery, spark.sql

SQLite

Embedded Database



```
import sqlite3
conn = sqlite3.connect("mydata.db")
cur = conn.cursor()
cur.execute("SELECT * FROM users;")
```

Use Cases

- Prototyping or small applications
- Embedded software or mobile apps
- Teaching and learning SQL
- Local data logging or analytics

Pros of SQLite

- ✓ Lightweight
- ✓ Easy setup
- ✓ Portable
- ✓ ACID-compliant
- ✓ Great for prototyping

Cons of SQLite

- ✗ Not for high concurrency
- ✗ Limited scalability
- ✗ Limited advanced features
- ✗ File-based



MYSQL

MySQL is an open-source relational database management system (RDBMS) known for its speed, reliability, and use in web and enterprise applications

INSTALLATION

- Install MySQL Server via the official installer
- Install the Python connector with the command:

```
pip install mysql-connector-python
```

- Connect to MySQL from Python using:

```
import mysql.connector  
conn = mysql.connector.connect(  
    host="localhost",  
    password="yourpassword")
```

BASIC OPERATIONS

- Create Table:
`CREATE TABLE users (...)
cursorexecute("CREATE TABLE users (...)`
- Insert Data:
`INSERT INTO users VALUES (...)
cursorexecute("INSERT INTO users VAL....)`
- Query Data:
`SELECT * FROM users
cursorexecute("SELECT * spers')`
- Update Data:
`UPDATE users SET name='Ali WHERE id=1
cursorexecute ("UPDATE users SET`
- Delete Data:
`DELETE FROM users WHERE id=1
cursorexecute ("DELETE FROM users")`

USE CASES

PostgreSQL (psycopg2)

1. Installation

Install PostgreSQL Server

- **Windows/macOS:** Download installer from <https://www.postgresql.org/download>
- **Linux (Ubuntu):**

```
sudo apt update sudo
```

```
apt install postgresql postgresql-contrib
```

Install Python Driver

```
pip install psycopg2-binary
```

Connect from Python

```
import psycopg2
```

```
conn = psycopg2.connect(host="localhost", database="mydb", user="myuser", password="mypassword")
cur = conn.cursor()
```

2. Basic Operations

Operation	SQL Command	Python Code
Create Table	CREATE TABLE users (id SERIAL, name TEXT)	cursor.execute(...)
Insert Row	INSERT INTO users (name) VALUES ('Alice')	cursor.execute(...) + conn.commit()
Select Rows	SELECT * FROM users	cursor.fetchall()
Update Row	UPDATE users SET name='Ali' WHERE id=	.execute()
Delete Row	DELETE FROM users WHERE id=1	Same as above + conn.commit()

ODBC Drivers

Generic Cross-Database Access

- Compatible SQL code for databases (SQL Server, Oracle, MySQL)
- Python drivers (`pyodbc`, `a-a`)-ofor supported databases

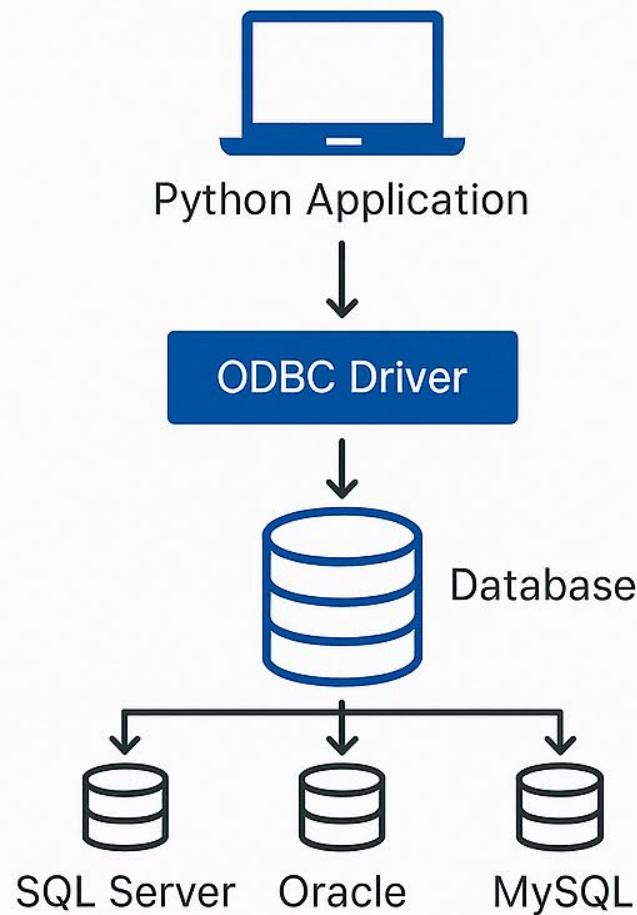
Setting up DSNs

- DSN stands: *Data Source Name* via ODBC Data Source Adm. files Configuration files

Advantages

- Cross-DB compatibility
- Centralized configuration
- Tool ecosystem support
- Integration with legacy apps

```
conn = pyodbc.connect("DSN=mydata-  
source;UID=user;PWD=pass")
```



Python odbc

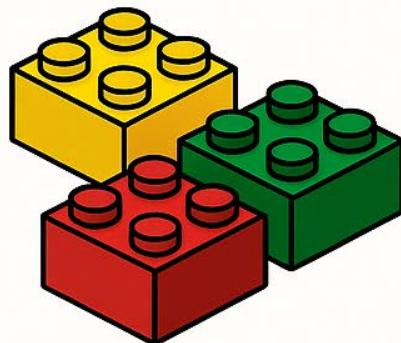
- import pyodbc
- conn = pyodbc.connect(
 - "Driver={SQL Server};"
 - "Server=localhost;"
 - "Database=MyDatabase;"
 - "Trusted_Connection=yes;"
 -)
- cursor = conn.cursor()
- cursor.execute("SELECT * FROM my_table")
- for row in cursor.fetchall():
 - print(row)
- conn.close()

What is SQLAlchemy?

SQLAlchemy is a Python tool that helps you talk to databases using Python instead of raw SQL.

SQLAlchemy Core

“Write SQL using Python”



Like using
LEGO blocks

SQLAlchemy ORM

*“Work with Python classes
instead of SQL”*



Like using pre-built
LEGO models

SQLAlchemy Core – Essentials

1 Engine Creation

The Engine is your link to the database.

```
from sqlalchemy import  
create_engine  
  
# For SQLITE (local file)  
engine = create_engine(sqlite:///data.db)  
  
# For PostgreSQL  
# engine = create_engine(  
#     sql://user:pas?  
#     @localhost/mybd"))
```

2 Connection Management

Use the engine to open a connection, do your work, and then close it.

```
with engine.connect()  
as conn:  
    # Do something  
    with conn
```

3 Executing SQL Statements

Once you have a connection, you can run SQL queries directly

```
from sqlalchemy import text  
  
with engine.connect() as conn:  
    result = conn.execute(text  
        ("SELECT * FROM users"))  
    for row in result:  
        print(row("name"))
```

SQLAlchemy ORM

Line	What it does
class User(Base)	Creates a Python class that will map to a SQL table
<code>__tablename__ = 'users'</code>	Sets the name of the table in the database
<code>id = Column(...)</code>	Defines a column in the table (and a class attribute)
<code>__repr__()</code>	A helper function to nicely print the object

SQLAlchemy Queries

❖ SQL Operation

1. Define Table

2. Insert

3. Select All

4. Select + Filter

5. Update

6. Delete

7. Commit

8. Schema Create

❖ SQLAlchemy Core (Manual)

```
Table('users', MetaData(),  
      Column(...))
```

```
insert(users).values(name  
                      ="Alice")
```

```
select(users)
```

```
select(users).where(users.  
                     c.name == "Alice")
```

```
update(users).where(...).va  
lues(...)
```

```
delete(users).where(...)
```

```
conn.commit()
```

```
metadata.create_all(engine)
```

❖ SQLAlchemy ORM (Object-oriented)

```
class User(Base): ...
```

```
session.add(User(name="  
                  Alice"))
```

```
session.query(User).all()
```

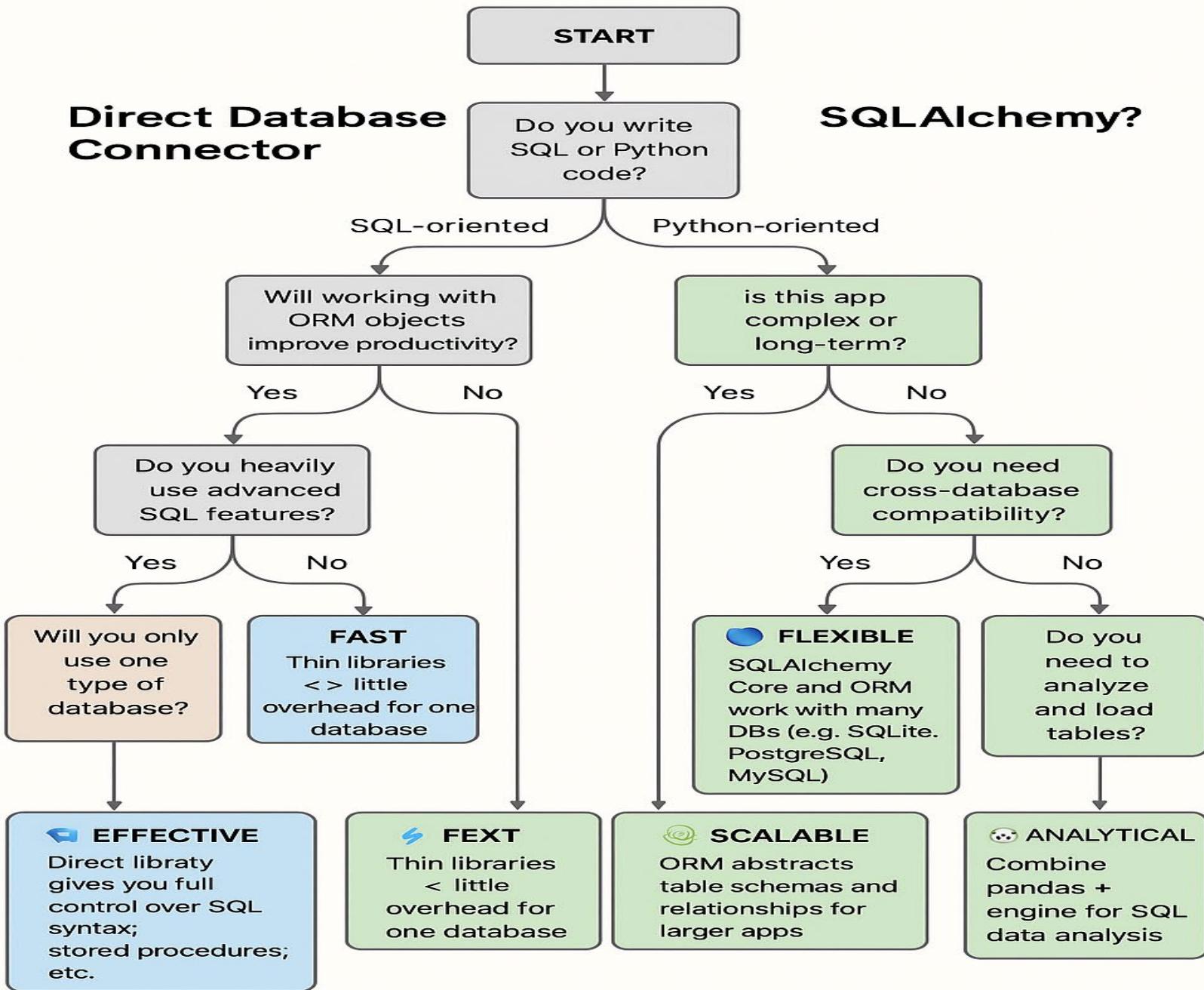
```
session.query(User).filter_  
by(name="Alice").first()
```

```
user.email = "..." +  
session.commit()
```

```
session.delete(user) +  
session.commit()
```

```
session.commit()
```

```
Base.metadata.create_all(  
                        engine)
```



SQLAlchemy: Metadata & Reflection

-  `MetaData()`: Container for schema (tables, columns, types)
-  Reflection: Auto-load tables from existing DB with `metadata.reflect(bind=engine)`

Connection strings

- `dialect+driver://username:password@host:port/database`

Component	Example	Description
dialect	sqlite, postgresql, mysql	Type of database you're connecting to
driver	psycopg2, pymysql	Python DBAPI/driver used for the connection
username	admin	Your database login username
password	mypassword	Your password (URL-encoded if needed)
host	localhost	Server address (e.g., 127.0.0.1, db.example.com)
port	5432, 3306	Port number (5432 for PostgreSQL, 3306 for MySQL)
database	mydb	Name of the database/schema you want to use

Database Connection Strings

`dialect+driver://username:password@host:port/atabase`



`sqlite:///path/to/`

`postgresql+psycopg2://user:password@localhost:5432/dbame`



`mysql+pymysql://user:password@host:port/dbname`

`mysql+pymysql //user:password@host:port/dbname`

Securing Database Connections



Credential Management

- Bad Practice: Hardcoding passwords

```
engine = create_engine(  
    "postgresql+psycopg2://admin:secret@cc1mdb")
```



Environment Variables

- Store credentials outside of code:

```
export DB_USER="admin"  
pwd = os.getenv("DB_PASS")  
import os  
user = os.getenv("DB_USER")  
pwd = os.getenv("DB_PASS")
```



Vault Services

- Store credentials securely in a dedicated service (e.g., HashiCorp Vault)

Basic Insert

```
stmt = insert(users).values (name="Ali", age=30)
with engine.connect() as conn:
    conn.execute(stmt)
    conn.commit()
```

- Uses SQLAlchemy Core
- Creates single row

ORM Insert

```
user = User(name="Ali")
age(30)
session.add(user)
session.commit()
```

- Uses SQLAlchemy ORM

Bulk Inserts

Faster inserts with multiple rows

ORM Bulk Insert

```
stmt = insert(users)
values = [
    {"name = "Sara",
     age: 25},
    {"name = "Zain",
     age: 38},
    {"name = "Ahmed"}
     age: 35]
```

1. Basic Update (ORM)

```
user = session.query(User).filter_by(id=1)
user.name = 'Ali'
session.commit
```

2. Raw SQL Update (SQLAlchemy Core)

```
from sqlalchemy import update
stmt = update(users).where(user.id == 1).values
(name="Ali")
with engine.begin() as conn:
    conn.execute(stmt)
```

3. Bulk Updates (ORM)

```
session.query(User).filter(User.age > 30)
.update({User.status: "Senior"},
synchronize_session=False)
.session.commit
```

USE CASES

- Small interactive changes
- Updating many rows

SQL DELETE Operations

Regular Delete

```
ORM user = session.query(User).get(1)
    session.delete(user)
    session.commit()
```

Delete related rows
in child tables

Cascading Delete

```
class Order(Base):
    __tablename__ = "orders"
    id = Column(Integer, primary_key=True)
    user_id = Column(Integer, ForeignKey("users.id"), CASCADE)
```

Mark rows as inactive
instead of removing
them

Soft Delete

```
Update stmt = update(users).where(users.id = 1)
    values(is_deleted=True)
Query active_users = session.query(User)
```

Mark rows as inactive
instead of removing
them

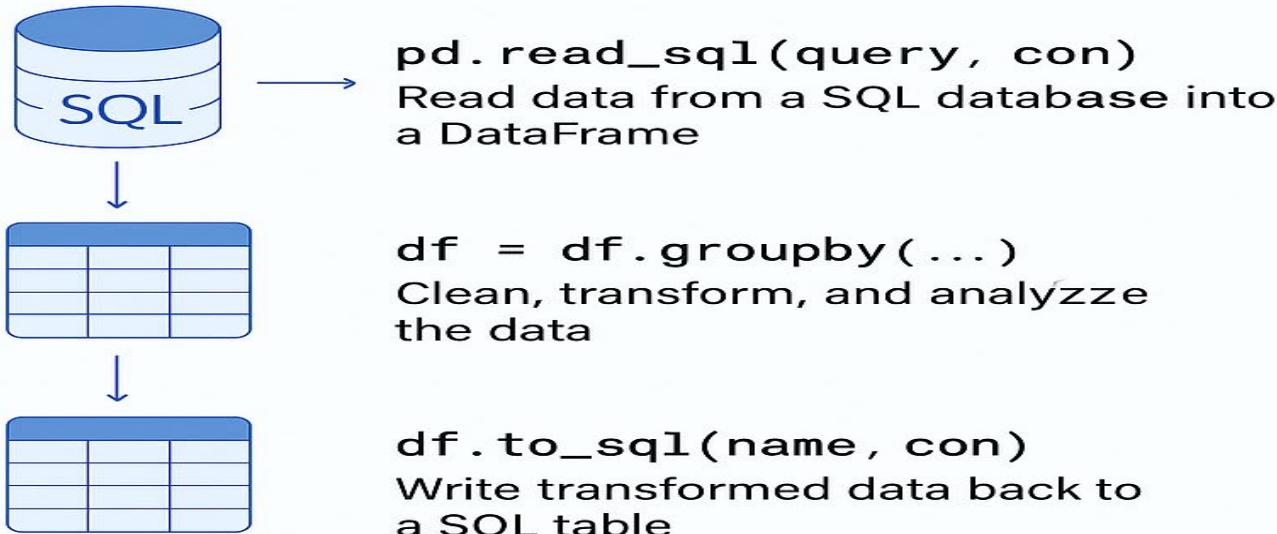
Why Pandas?

DataFrames and SQL

What is Pandas?

- A powerful Python library for data analysis
- Provides DataFrame: a table-like data structure
- Seamlessly interacts with SQL databases

Example Workflow



Key Benefits

- Simplifies data manipulation and analysis
- Leverages SQL for data storage and retrieval
- Ideal for ETL (Extract, Transform, Load) tasks

read_sql_query() vs read_sql_table()

Feature	read_sql_query()	read_sql_table()
Purpose	Run SQL queries	Load entire table
Input	SQL string (e.g. SELECT...)	Table name (string)
Custom SQL Support	<input type="checkbox"/> Full SQL support	<input type="checkbox"/> Only full table
Requires SQLAlchemy	<input type="checkbox"/> Not required	<input type="checkbox"/> Required
Use Case	Filters, joins, analysis	Quick full table load

Chunked Loading & Memory Management in Pandas

□ Chunked Loading Example

```
chunksize = 10000
for chunk in pd.read_sql_query("SELECT * FROM big_table",
                               con=engine, chunksize=chunksize):
    process(chunk)
```

□ Memory Management Techniques

- Use `chunksize=` to process large datasets in batches
- Filter data using SQL WHERE clauses before loading
- Select only required columns (`SELECT col1, col2`)
- Use `del df` and `gc.collect()` to free memory manually
- Convert `float64` to `float32` to reduce memory usage
- Work with compressed formats like GZIP or Parquet

Using `to_sql()` in Pandas & Handling Schema Differences

□ Basic Syntax

```
df.to_sql(  
    name="table_name",  
    con=engine,  
    if_exists="append", # or "replace", "fail"  
    index=False  
)
```

⊗ **if_exists** Options

- "fail" → Raise error if table exists
- "replace" → Drop & recreate table
- "append" → Add to table (schemas must match)

□ Schema Handling Techniques

- Align DataFrame columns before writing
- Inspect schema with `inspect(engine).get_columns()`
- Use `dtype` to define column types (e.g., Integer, String)
- Use 'replace' carefully — it drops the table!

Case Study: Retail Sales Analysis

- Data acquisition
- SQL queries
- Data analysis steps

Case Study: Retail Sales Analysis

□ Data Acquisition

- Source: SQL database (PostgreSQL / MySQL / SQLite)
- Tables: customers, orders, order_items, products
- Loaded via Pandas `read_sql_query()` or SQLAlchemy

□ SQL Queries

- Total revenue by region using JOIN + GROUP BY
- Top-selling product using SUM and ORDER BY
- Filter by timeframe or category

□ Data Analysis Steps

- Calculate revenue: `df['revenue'] = df['quantity'] * df['price']`
- Group by product/category/region using `groupby()`
- Visualize with `matplotlib` or `seaborn`
- Export summaries to Excel/CSV if needed

Week 11: Exploratory Data Analysis (EDA)

- Course: Business Data Analysis (MGT-174)
- Week 11 | Exploratory Data Analysis
- Instructor: Dr. Muhammad Usman Bhutta

EDA?

Exploratory data analysis (EDA) is used by data scientists to analyze and investigate data sets and summarize their main characteristics, often employing data visualization methods.

EDA helps determine how best to manipulate data sources to get the answers you need, making it easier for data scientists to discover patterns, spot anomalies, test a hypothesis, or check assumptions.

EDA?

EDA is primarily used to see what data can reveal beyond the formal hypothesis testing task and provides a better understanding of data set variables and the relationships between them.

It can help determine if the statistical techniques you are considering for data analysis are appropriate.

EDA in Business Context

- Business decisions depend heavily on data insights.
- EDA allows managers to understand trends, outliers, and customer behavior from the data.

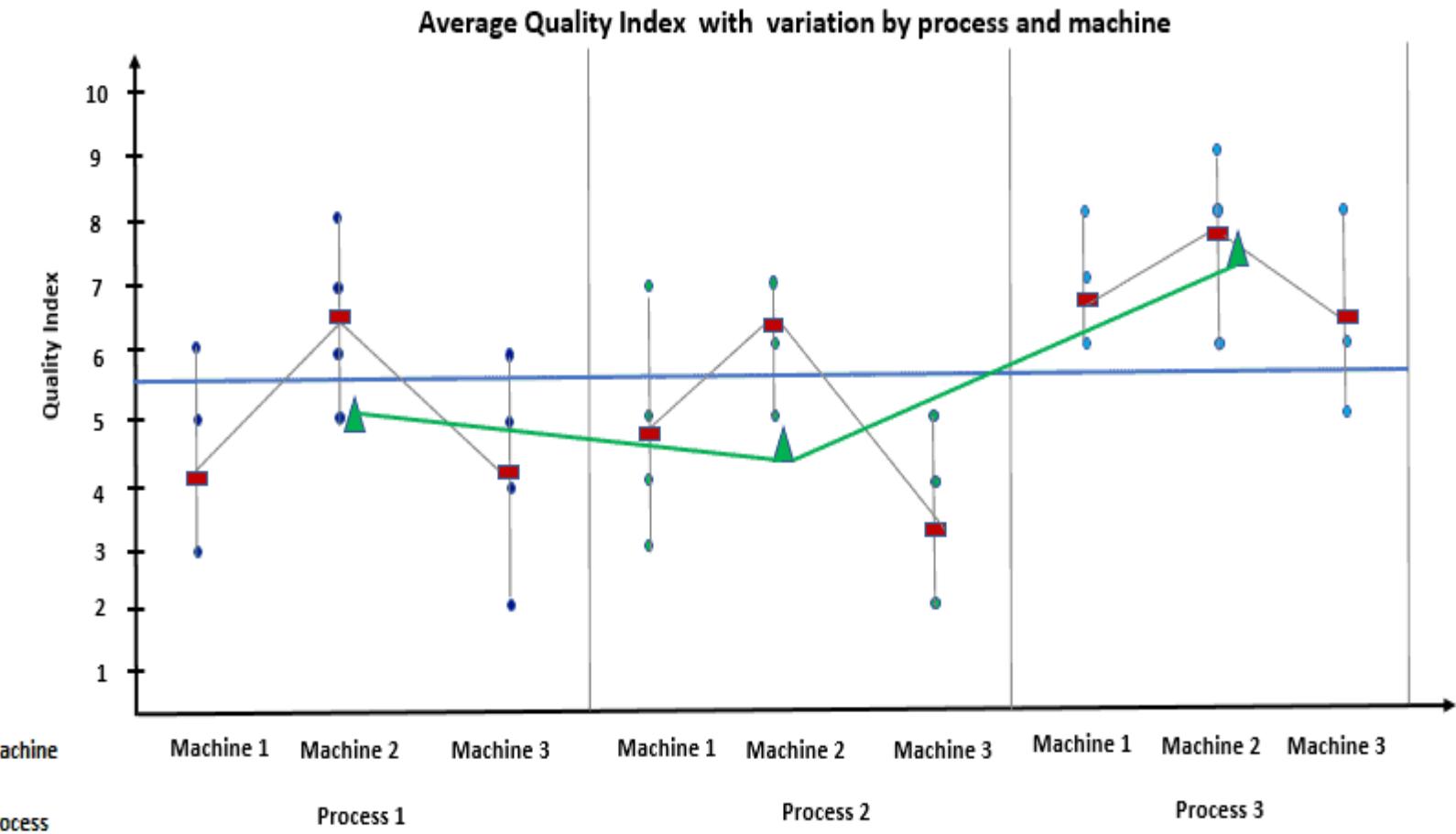
Core Objectives of EDA

- Understand data structure and distribution.
- Detect anomalies and outliers.
- Test assumptions and generate hypotheses.
- Select features for modeling.
 - Using correlations analysis, variance, missing values.

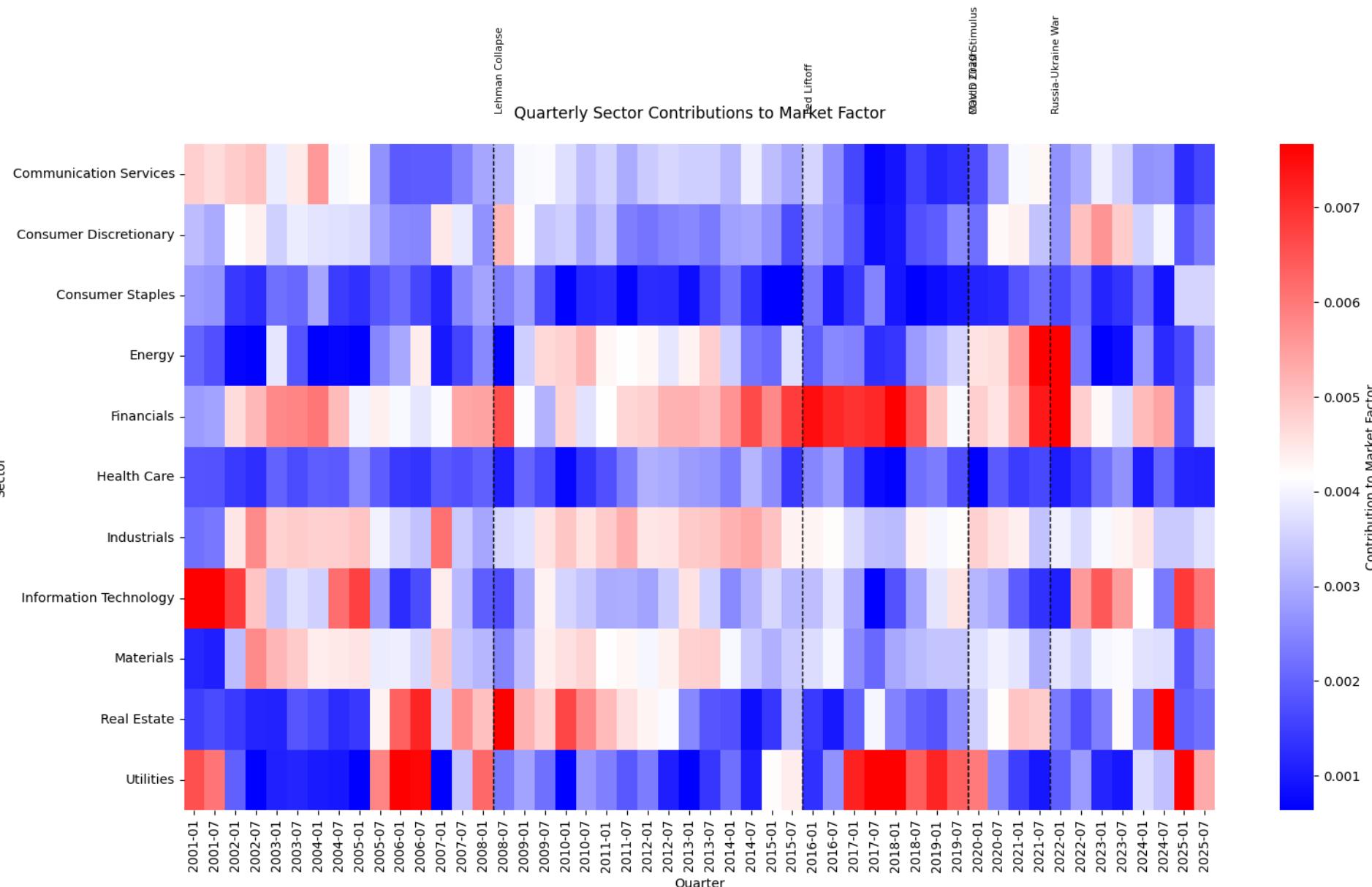
Stem-and-Leaf Plot

Stem	Leaf
1	0 2 4
2	1 3 5 8
3	0 3 3 8
4	1 2 4 7
5	0 3 5 7
6	0

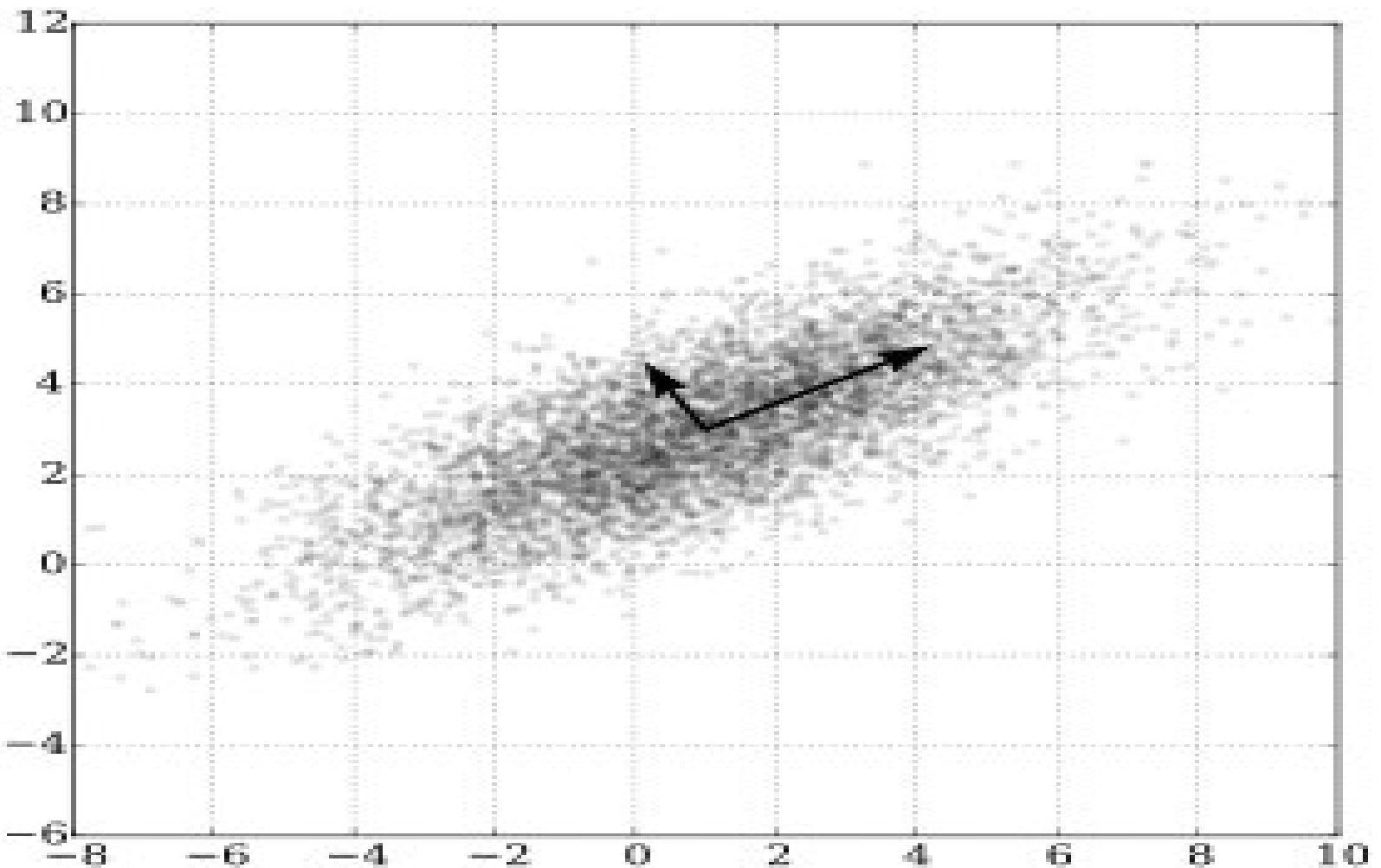
Multi-Vari Chart



Heatmaps



PCA



Median polish

- Extract :
 - Overall median
 - Row median
 - Column median
- Continue until residuals stabilize-close to zero

Multivariate Analysis – Correlation

- Correlation measures linear relationships between numeric variables.
- Use `df.corr()` or `sns.heatmap()` to visualize correlation matrix.

K-mean clustering

- - What is Clustering?
- - Introduction to K-Means
- - Algorithm Steps
- - Example Use Cases

What is Clustering?

- - Grouping data points based on similarity
 - KMEANS clustering

K-Means Clustering

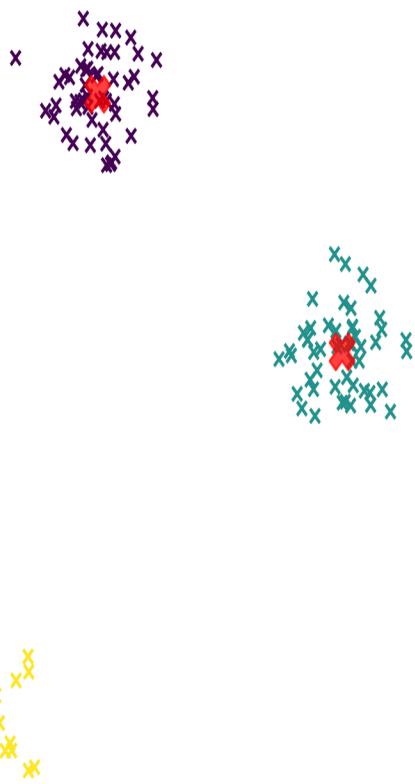
- - Partitional clustering technique
- - Divides data into K non-overlapping clusters
- - Each point belongs to the nearest centroid

How K-Means Works

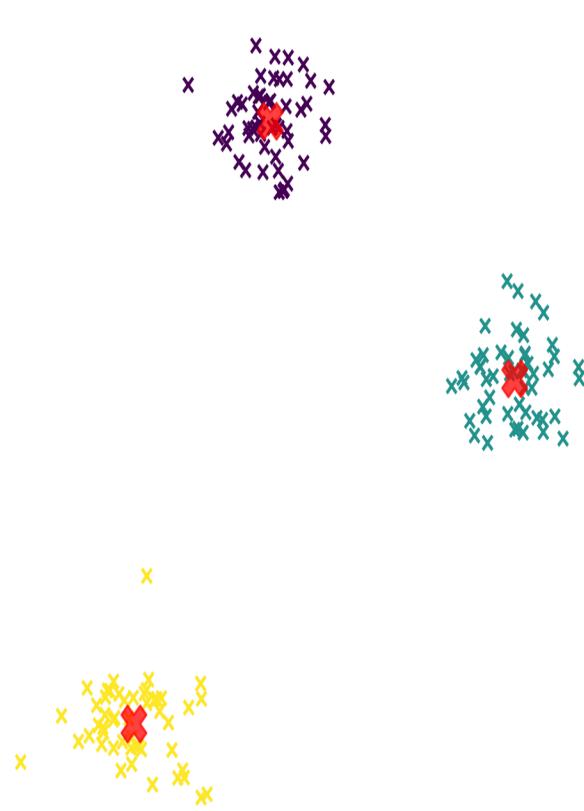
- 1. Choose K
- 2. Initialize centroids
- 3. Assign points to nearest centroid
- 4. Recompute centroids
- 5. Repeat until convergence

Visual Example

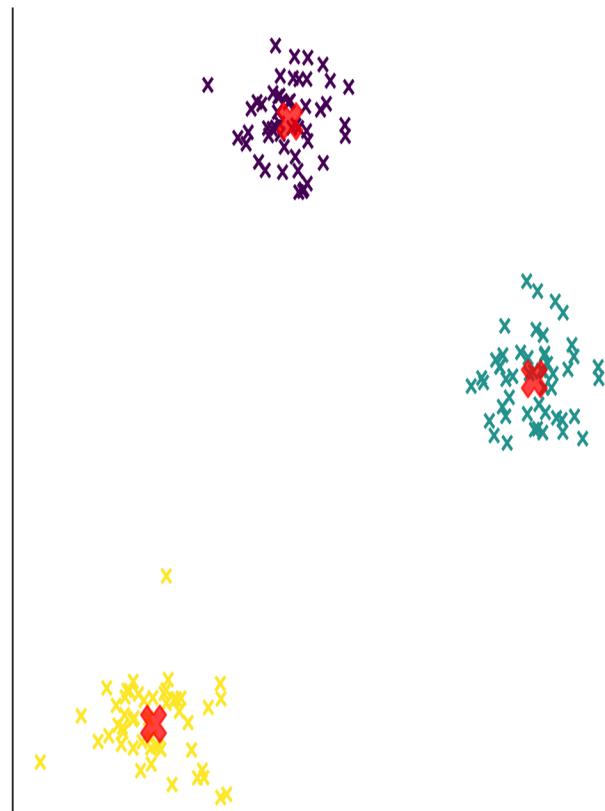
Iteration 1



Iteration 2



Iteration 3



Example Use Cases

- - Customer segmentation
- - Image compression
- - Market basket analysis

Week 12 - Introduction to Machine Learning

- Business Data Analysis
- Instructor: Dr Muhammad Usman Bhutta

Section 1: Introduction to Machine Learning

	Machine Learning	Statistics	Computer Science
Objective	Focuses on learning from data to make predictions or decisions without being explicitly programmed. It prioritizes prediction accuracy and generalizability.	Aims to infer properties of an underlying distribution from a data sample. It emphasizes understanding and interpreting data and probabilistic models.	Focuses on the creation and application of algorithms to manipulate, store, and communicate digital information.
Methodologies	Typically uses complex models (like neural networks) and large amounts of data to train models for prediction. Utilizes both supervised and unsupervised learning methods.	Often employs simpler, more interpretable models. Focuses on hypothesis testing, experimental design, estimation, and mathematical analysis.	Involves algorithm design, data structures, computation theory, computer architecture, software development, and more.
Validation	Measures model performance through methods like cross-validation and seeks to improve generalization to unseen data.	Validates models using methods such as confidence intervals, p-values, and hypothesis tests to quantify uncertainty.	Uses formal methods for verifying correctness, analyzing computational complexity, and proving algorithmic bounds.
Primary Concern	Creating models that can learn from and make decisions or predictions based on data.	Drawing valid conclusions and quantifying uncertainty about observed data and underlying distributions.	Creating efficient algorithms and data structures to solve computational problems.

What is Machine Learning? (Detailed Definition)

- Machine Learning (ML) is a subset of artificial intelligence that enables systems to learn from data, identify patterns, and make decisions with minimal human intervention.

Key Components of ML

- - Data: Raw information to train the model
- - Model: Learns the relationship between input and output
- - Predictions: Outputs generated for new, unseen data

Pattern Recognition and Automation

- ML automates the process of finding patterns in large datasets, allowing for
 - Scalability: Models can handle vast volumes of data without manual intervention.
 - Efficiency: Replaces repetitive decision-making tasks with automated systems.
 - Consistency: ML systems apply the same logic across all data, reducing human error.

Illustration: ML Pipeline

- [Input Data] → [Preprocessing] → [ML Model (Training)] → [Predictions]
- The model learns patterns during training and applies them during prediction.

ML vs Traditional Programming

Traditional Programming

Rules + Data → Output

Explicit instructions are coded

Limited adaptability

Example: If-else rules for spam

Manual updates needed

Machine Learning

Data + Output → Model (learned rules)

Model learns patterns from data

Adapts to new data automatically

Example: Train spam filter on email data

Self-improving with more data

- In traditional programming, developers write all the logic.
- In ML, you feed the system data and examples of correct output; it “learns” the logic itself.

Example

- Input: Email text (Data)
- Labels: Spam / Not Spam (Output)
- ↓
- Algorithm learns word patterns → builds a Model
- ↓
- New Emails → Classified as Spam or Not Spam

Importance of ML in Business Analytics

- Why Businesses Use ML
 - Make data-driven decisions faster and more accurately
 - Discover hidden patterns in customer, sales, or market data
 - Improve forecasting and predictions
-  Common Business Applications
 -  Sales Forecasting – Predict demand and revenue
 -  Customer Segmentation – Group customers based on behavior
 -  Churn Prediction – Identify customers likely to leave
 -  Fraud Detection – Spot suspicious financial transactions
 -  Inventory Optimization – Manage stock based on trends

Real-World Applications of ML

-  **Email Filtering** – Detect spam and categorize messages
-  **Recommendation Systems** – Netflix, YouTube, and Amazon suggestions
-  **Self-Driving Cars** – Perception, prediction, and control using ML
-  **Healthcare Diagnostics** – Predict diseases from medical images
-  **Fraud Detection** – Spot abnormal patterns in financial transactions
-  **Customer Service Chatbots** – Automate responses using Natural Language Processing
-  **Stock Market Predictions** – Analyze historical trends and indicators

ML in Python Ecosystem

Why Python for Machine Learning?

- Simple syntax, easy to learn
- Massive open-source community support
- Rich ecosystem of specialized libraries

Popular ML Libraries in Python

- **NumPy** – Efficient numerical computations
- **Pandas** – Data manipulation and analysis
- **Matplotlib / Seaborn** – Data visualization
- **scikit-learn** – Core machine learning toolkit
- **TensorFlow / PyTorch** – Deep learning frameworks
- **Statsmodels** – Statistical modeling

Key ML Libraries (scikit-learn, pandas, numpy)

Library	Role	Basic Syntax
NumPy	Numerical arrays & math	<code>import numpy as np np.array([...])</code>
Pandas	Data manipulation	<code>import pandas as pd df = pd.read_csv()</code>
Matplotlib	Visualization (basic)	<code>import matplotlib.pyplot as plt plt.plot()</code>
Seaborn	Visualization (advanced, stats)	<code>import seaborn as sns sns.histplot()</code>
scikit-learn	Machine learning models & metrics	<code>from sklearn.linear_model import LinearRegression model.fit(X, y)</code>

Types of Machine Learning (Overview)

Type	What It Does	Examples
Supervised Learning	Learns from labeled data (input + known output)	Email spam detection, price prediction
Unsupervised Learning	Finds patterns in unlabeled data	Customer segmentation, Fraud Detection
Reinforcement Learning	Learns by trial and error to maximize rewards	Autonomous Driving, dynamic pricing

Supervised vs Unsupervised Learning

Aspect	Supervised Learning	Unsupervised Learning
Data	Labeled (input + correct output)	Unlabeled (only input features)
Goal	Predict outcomes based on past examples	Discover hidden patterns or groupings
Approach	Label-based training	Structure-discovery methods
Examples	Email spam detection, price prediction	Customer segmentation, anomaly detection
Common Algorithms	Linear regression, decision trees, SVM	K-Means, PCA, hierarchical clustering

Supervised Learning

- Trained using labeled examples
- Desired output is *known*
- Methods include classification, regression, etc.
- Uses patterns to predict the values of the label on additional unlabeled data
- Algorithms:
 - Linear regression
 - Logistic regression
 - Decision Trees and Random Forests
 - Neural Networks

Definition and Use Cases

Supervised learning involves training a model using **labeled data**, where both input features and correct outputs are known.

Real-World Example: Sales Forecasting

- **Inputs (X)**: Advertising spend, season, region, number of salespeople
- **Label (y)**: Units sold or revenue
- The model learns the relationship between inputs and target sales figures.

Model Goal:

Predict future sales based on new input conditions.

Other Examples:

- Predicting housing prices
- Classifying emails as spam or not spam
- Diagnosing diseases from symptoms

Classification vs Regression

Task	Classification	Regression
Output Type	Discrete categories (labels)	Continuous values (real numbers)
Goal	Assign input to a category	Predict a numeric quantity
Examples	Spam vs Non-Spam, Disease A/B/C	Predict house price, sales revenue
Algorithms	Logistic Regression, Decision Trees	Linear Regression, Random Forest Regressor

Input (Features) and Output (Labels)

 Input = Features (X) The measurable attributes used to make predictions

Can be numerical (e.g., age, income) or categorical (e.g., gender, region)

Example: $X = [\text{Advertising Budget}, \text{Season}, \text{Region}]$

 Output = Label (y) The value we want the model to predict

Can be a class (for classification) or a number (for regression)

Example: $y = \text{Sales Revenue}$ or $y = \text{Spam} / \text{Not Spam}$

 The Goal: Learn a mapping function: $f(X) \rightarrow y$

Train/Test Split



Why Split the Data?

To evaluate how well a machine learning model generalizes to **unseen data**.



Typical Split Ratios:

- 80% for training
- 20% for testing
(sometimes 70/30 or 60/40)



Training Set:

- Used to fit the model
- Model learns the patterns



Testing Set:

- Used to evaluate model accuracy

Model Training: Concept



What is Model Training?

Training is the process of teaching a model to **find patterns** in labeled data (features and labels).



How It Works:

- The model takes inputs (features) and tries to predict outputs (labels)
- It compares predictions to actual values
- It adjusts internal parameters to reduce error



Objective:

Minimize the difference between predicted and actual outputs

→ This is called **loss** or **error (Minimization)**

Model Evaluation Metrics

-  Why Evaluate a Model?
- To measure how well the model performs on unseen data.
-  For Classification Tasks:
 - Accuracy: % of correct predictions
 - Precision: How many predicted positives were actually positive
 - Recall: How many actual positives were correctly predicted
 - F1 Score: Harmonic mean of precision and recall

Introduction to Linear Regression

- Simple regression formula, interpretation of coefficients.
- $y = \beta_0 + \beta_1 x + \epsilon$
- y : the output or dependent variable
- x : the input or independent variable
- β_0 : the intercept (value of y when $x=0$)
- β_1 : the slope or coefficient (how much y changes for a unit change in x)
- ϵ : the error term (accounts for variability not captured by the model)

Hands-on: Linear Regression in Python

- Using scikit-learn to fit and predict.

Classification: Logistic Regression Overview

- Log-odds, sigmoid function, binary outcomes.

Hands-on: Logistic Regression in Python

- Example using
`sklearn.linear_model.LogisticRegression.`

Section 3: Unsupervised Learning

Unsupervised Learning — an important branch of machine learning that works **without labeled data**.

Unlike supervised learning, where we provide both inputs and outputs (like predicting price or classifying spam), in unsupervised learning we only give the model **input data** — and ask it to find **structure or patterns** on its own.

Common goals in unsupervised learning include:

- **Clustering**: grouping similar data points (e.g., customer segmentation).
- **Dimensionality Reduction**: simplifying data while preserving structure (e.g., PCA).
- **Anomaly Detection**: identifying unusual data points (e.g., fraud or network intrusion).

Clustering and Dimensionality Reduction

◆ Clustering:

- The goal is to **group similar data points** together.
- Each group, or *cluster*, ideally contains items that are more similar to each other than to items in other clusters.
- Clustering is often used in:
 - **Customer segmentation**
 - **Market basket analysis**
 - **Anomaly detection**

◆ Dimensionality Reduction:

- Here, the aim is to **reduce the number of input features** while retaining the most important structure or variance in the data.
- This is especially useful when dealing with **high-dimensional datasets**.
- Common applications include:
 - **Data visualization** (e.g., PCA to 2D or 3D)
 - **Noise reduction**
 - **Speeding up machine learning algorithms**

⟳ Key Difference:

- **Clustering** finds hidden groups or structure.
- **Dimensionality reduction** simplifies the data representation without losing too much information.

K-Means Clustering: Basic Idea

- Choose the number of clusters, k , beforehand.
- Randomly initialize k centroids (central points for each cluster).
- Assign each data point to the nearest centroid, using Euclidean distance.
- Recalculate the centroids as the average position of all points in each cluster.
- Repeat steps 3 and 4 until the centroids stop moving significantly — that's when the algorithm has converged.

Hands-on: K-Means in Python

- Using `sklearn.cluster.KMeans`.
- `import numpy as np`
- `import matplotlib.pyplot as plt`
- `from sklearn.cluster import KMeans`
- `from sklearn.datasets import make_blobs`
- **# Generate synthetic 2D data**
- `X, _ = make_blobs(n_samples=300, centers=3, random_state=42)`
- **# Fit K-Means model**
- `kmeans = KMeans(n_clusters=3, random_state=42)`
- `kmeans.fit(X)`
- **# Get cluster assignments and centroids**
- `labels = kmeans.labels_`
- `centroids = kmeans.cluster_centers_`

PCA: Basic Intuition

- Reducing dimensionality while preserving variance.

What does PCA actually do?

- Imagine you have data with many variables (e.g., 10, 100, or even 1,000).
- Some of those variables might be **redundant or correlated**.
- PCA transforms the data into a new coordinate system:
 - Each new coordinate (called a **principal component**) is a **linear combination** of the original features.
 - The first principal component captures the **most variance**, the second captures the next most, and so on.

Hands-on: PCA with Python

- **Using `sklearn.decomposition.PCA`.**
- `data = load_iris()`
- `X = data.data`
- `y = data.target`
- `# Standardize the features (important before PCA)`
- `X_scaled = StandardScaler().fit_transform(X)`
- `# Apply PCA: reduce to 2 components`
- `pca = PCA(n_components=2)`
- `X_pca = pca.fit_transform(X_scaled)`
- `# Print explained variance ratio`
- `print("Explained variance ratio:",
pca.explained_variance_ratio_)`

Use Cases of Unsupervised Learning

- Market segmentation, anomaly detection,

Data Cleaning & Feature Engineering

- Imputing, encoding, creating new features.

Scaling and Normalization

- StandardScaler, MinMaxScaler examples.

Cross-Validation Basics

- K-fold strategy for generalization.

Model Evaluation Revisited

- Confusion matrix, classification report.

Hyperparameter Tuning

- GridSearchCV: process and example.

Overfitting and Underfitting

- Bias-variance tradeoff explained visually.

Section 5: Lab Orientation & Assignment

Q&A and Wrap-Up

- Open discussion, address confusion.