

# DATA ANALYTICS LEARNING MODULE

## Day 1

### DATA ANALYTICS INTRODUCTION

#### What is Data?

Data is raw facts, figures, or information collected from various sources. It can be numbers, text, images, videos, or any other format that can be stored and processed.

#### Types of Data

There are two main types of data:

##### 1. Structured Data (Well-Organized, Easy to Store)

- Data is arranged in rows and columns like a table.
- Easy to store, search, and analyze using databases like MySQL, Excel, or Google Sheets.

##### Example of Structured Data (Supermarket Sales Table):

Date	Product	Quantity Sold	Price
01-Feb-24	Laptop	5	Rs.40000
01-Feb-24	Mouse	15	Rs.1000
02-Feb-24	Keyboard	10	Rs. 2500

This data is easy to analyze using SQL, Excel, or Python.

## 2. Unstructured Data (Unorganized, Difficult to Analyze)

- Data that does not have a fixed format (like images, videos, emails, chat messages, etc.).
- Cannot be stored in tables easily but needs tools like AI, Machine Learning, and Big Data technologies to process.

### Example of Unstructured Data:

- Emails sent by customers with feedback.
- Social media comments and tweets about a product.
- CCTV footage in a mall.
- WhatsApp messages and voice notes.

Unstructured data needs advanced tools (AI, NLP, Image Recognition) for analysis.

## Categories of Data Based on Measurement

### 1. Quantitative Data (Numbers & Measurable)

Numerical data that can be counted or measured.

#### Examples:

- Supermarket: "500 customers visited the store today."
- Health: "A person's body temperature is 98.6°F."
- Finance: "Company profit is \$1.2 million."

Used in reports, statistics, and financial analysis.

### 2. Qualitative Data (Descriptive & Non-Measurable)

Descriptive data that describes qualities, but cannot be measured with numbers.

### **Examples:**

- Customer Feedback: "The product is amazing!"
- Movie Review: "This movie is very interesting."
- Survey Response: "The service was excellent."

Used in sentiment analysis, market research, and customer feedback analysis.

### **How is Data Used in Real Life?**

- Amazon: Uses customer purchase history data to recommend products.
- Netflix: Uses watch history data to suggest movies.
- Banks: Use transaction data to detect fraud.
- Hospitals: Use patient health data to provide better treatments.

### **Summary**

Type of Data	Description	Example
<b>Structured Data</b>	Organized in rows & columns	Sales data in Excel
<b>Unstructured Data</b>	No fixed format	Social media comments, Images
<b>Quantitative Data</b>	Numbers & measurable	1000 customers, \$500 sales
<b>Qualitative Data</b>	Descriptive & not measurable	"Great product!", "Nice service"

### **What is Data Analytics?**

Data Analytics is the process of analyzing raw data to find patterns, trends, and insights that help in decision-making.

Simply, Data Analytics = Data + Analysis (Understanding information and making smart decisions!)

## What is Raw Data and Insight?

### 1. Raw Data (Unprocessed Data)

Raw Data is the original, unorganized, and unprocessed information collected from various sources. It does not give clear meaning until analyzed.

#### Example:

Imagine a supermarket collects sales data daily. Their raw data might look like this:

Date	Product	Quantity Sold	Price	Customer Location
01-Feb-24	Laptop	5	\$1000	New York
01-Feb-24	Mouse	15	\$20	California
02-Feb-24	Keyboard	10	\$50	Texas
02-Feb-24	Laptop	3	\$1000	Florida

This is raw data – it's just numbers and text without any meaningful conclusion.

### 2. Insight (Meaningful Information from Data)

Insight is the useful conclusion you get after analyzing raw data.

It helps businesses make smart decisions.

#### Example of Insights from the above raw data:

- Laptops sell more in New York than in Florida → Maybe customers in New York need laptops more.
- Mouse sales are high in California → The store can increase stock in that region.
- Keyboards are also selling well → Running a Keyboard + Mouse combo offer may increase sales.

Insights help businesses take the right actions based on data!

## Summary

Feature	Raw Data	Insight
Definition	Unprocessed, unorganized information	Useful conclusions from data
Example	List of all product sales in a store	"Laptops are in high demand in New York"
Usefulness	Cannot make decisions	Helps in decision-making

### Example 1: Supermarket Sales Analysis

#### Situation:

A supermarket owner wants to know:

- Which products are selling the most?
- Which month has the highest sales?
- Which offers are attracting customers?

#### How Data Analytics Helps?

- Step 1: Collect sales data (e.g., sales from Jan to Dec 2024).
- Step 2: Analyze data using Excel / SQL / Python/ Power BI.
- Step 3: Create graphs & charts to identify trends.
- Step 4: Make business decisions based on insights.
  
- Findings: Chocolates have higher sales in December → Offer discounts for Christmas!
- Action: Increase promotions on chocolates → Boost sales!

This is how Data Analytics helps businesses!

## Example 2: Netflix Movie Recommendations

### Situation:

How does Netflix suggest movies you might like?

### How Data Analytics Helps?

- Netflix collects your data – What movies & genres do you watch?
- Finds patterns – If you watch action movies, similar ones will be suggested.
- Uses Machine Learning Algorithms – Compares your taste with other users.
- Findings: You watched a lot of Sci-Fi movies → Netflix suggests more Sci-Fi content!
- Action: Personalized movie recommendations

This is Predictive Data Analytics in action!

## Example 3: Amazon Product Pricing & Demand Analysis

### Situation:

Why do the same products on Amazon have different prices at different times?

### How Data Analytics Helps?

- Step 1: Amazon analyzes sales data (What time do people buy the most?)
- Step 2: Compares competitor prices
- Step 3: Checks demand vs supply
- Step 4: Adjusts prices dynamically
- Findings: More people buy electronics during festivals like Diwali & Christmas → Prices increase!
- Action: Dynamic Pricing – When demand is high, the price increases!

This is Business Data Analytics in action!

## Types of Data Analytics

There are 4 main types of Data Analytics:

### 1. Descriptive Analytics (What happened?)

Used for past analysis.

**Example:** A company checks last year's sales report to see which product performed best.

### 2. Diagnostic Analytics (Why did it happen?)

Finds the reason behind trends.

**Example:** If sales dropped last month, Diagnostic Analytics helps find the cause (e.g., customer reviews, competitor analysis).

### 3. Predictive Analytics (What will happen next?)

Forecasts future trends based on data.

**Example:** Netflix predicts which shows you will like based on your watch history.

### 4. Prescriptive Analytics (What should we do about it?)

Recommends the best action.

**Example:** Amazon suggests discounts on slow-selling products to clear stock.

## Why Learn Data Analytics?

- ✓ Businesses make data-driven decisions (Marketing, Sales, Finance, Healthcare).
- ✓ Real-world applications in daily life (Shopping, Movies, Banking, Sports).
- ✓ High-demand career – Data Analysts, Data Scientists, BI Analysts.

## Tools and Technologies in Data Analysis

Data analysis requires different tools and technologies to collect, clean, analyze, visualize, and interpret data. Let's explore the key tools used in each stage of data analysis.

### Data Collection Tools (To Gather Raw Data)

These tools help in collecting data from different sources like databases, APIs, websites, and sensors.

Tool	Usage	Example
SQL (MySQL, PostgreSQL, SQL Server)	Retrieve data from structured databases	"Get all sales data from the last 6 months."
Web Scraping (BeautifulSoup, Scrapy, Selenium)	Extract data from websites	"Scrape product prices from Amazon."
APIs (REST, GraphQL, Postman)	Get data from online services	"Fetch live stock market prices."
Google Forms & Excel	Manual data collection	"Survey responses about customer satisfaction."

### Data Cleaning Tools (To Remove Errors & Prepare Data)

Raw data often has missing values, duplicates, and errors. These tools help clean and preprocess the data.

Tool	Usage	Example
Python (Pandas, NumPy)	Handle missing values, duplicates, and format data	"Remove empty rows from an Excel sheet."
Excel (Power Query, Filters, Pivot Tables)	Clean and filter data manually	"Fix incorrect product prices in a spreadsheet."

OpenRefine	Detect and correct data inconsistencies	"Standardize product categories like 'Laptop' & 'laptops'."
------------	---	---

### **Data Analysis & Statistical Tools (To Find Patterns and Trends)**

These tools help analyze data using statistics, machine learning, and predictive models.

Tool	Usage	Example
Python (Pandas, NumPy, SciPy, Statsmodels)	Perform mathematical operations & statistical analysis	"Find average sales per month."
R (ggplot2, dplyr, tidyr)	Statistical modeling & hypothesis testing	"Check if weekend sales are higher than weekdays."
Excel (Pivot Tables, Formulas, Solver)	Basic data analysis without coding	"Summarize revenue from different regions."
SPSS & SAS	Advanced statistical analysis for business	"Analyze customer survey data for trends."

### **Data Visualization Tools (To Create Charts and Graphs)**

These tools convert complex data into easy-to-understand charts, graphs, and dashboards.

Tool	Usage	Example
Tableau	Drag-and-drop visual analytics	"Create an interactive sales dashboard."
Power BI	Microsoft's business intelligence tool	"Show monthly profit trends in a company report."
Matplotlib & Seaborn (Python)	Create custom charts using code	"Plot sales growth using a bar chart."
Excel Charts	Quick visualization with basic charts	"Create a pie chart for product sales distribution."

## Big Data & Cloud Technologies (For Large-Scale Data Processing)

These tools handle huge volumes of data (Big Data) and support cloud-based storage & analysis.

Tool	Usage	Example
Hadoop & Spark	Process large-scale unstructured data	"Analyze billions of transactions in a bank."
Google BigQuery	Cloud-based data warehouse	"Run SQL queries on terabytes of customer data."
AWS, Azure, Google Cloud	Cloud storage & computing	"Store and analyze IoT sensor data in real-time."

## Machine Learning & AI for Advanced Data Analysis

These tools help predict future trends, detect fraud, and automate decisions.

Tool	Usage	Example
Scikit-Learn (Python)	Machine learning models	"Predict if a customer will buy a product."
TensorFlow & PyTorch	Deep learning & AI models	"Detect customer emotions from social media comments."
AutoML (Google, Microsoft, H2O.ai)	Automated machine learning	"Find the best model for predicting sales."

## Summary Table: Data Analytics Tools by Category

Stage	Tools
Data Collection	SQL, Web Scraping, APIs, Google Forms, Excel
Data Cleaning	Python (Pandas, NumPy), Excel, OpenRefine
Data Analysis	Python (Pandas, NumPy, Statsmodels), R, Excel, SPSS
Data Visualization	Tableau, Power BI, Matplotlib, Seaborn, Excel Charts
Big Data & Cloud	Hadoop, Spark, Google BigQuery, AWS, Azure
Machine Learning	Scikit-Learn, TensorFlow, AutoML, PyTorch

## Real-Life Example of How These Tools Work Together

Imagine you are working for **Amazon** and want to analyze customer purchases:

1. Collect Data → Use SQL to get sales data from the Amazon database.
2. Clean Data → Use Python (Pandas) to remove duplicate orders.
3. Analyze Data → Use Excel Pivot Tables to find top-selling products.
4. Visualize Trends → Use PowerBI to create a bar chart of sales by category.
5. Predict Future Sales → Use Scikit-Learn to forecast demand for next month.

## How Does Data Analytics Work?

Data analytics follows a structured process to collect, clean, analyze, visualize, and interpret data for better decision-making. Let's break down the Data Analytics Workflow step by step with real-world examples.

### Data Collection (Gathering Raw Data)

#### What happens?

- Raw data is collected from different sources like databases, websites, IoT sensors, social media, APIs, and surveys.

#### Real-Life Example

E-commerce (Amazon): Collects user data on products viewed, items added to cart, and purchases made.

Healthcare (Hospitals): Gathers patient details like heart rate, temperature, and medical history from health monitoring devices.

#### Tools Used:

SQL, Web Scraping (BeautifulSoup, Scrapy), APIs (Postman), Google Forms, Excel

## Data Cleaning (Preparing the Data)

### What happens?

- Raw data is often messy (contains duplicates, missing values, errors, or inconsistent formats).
- Cleaning ensures data is correct, complete, and structured before analysis.

### Real-Life Example

Amazon: Removes duplicate orders, corrects product category mismatches, and fills missing shipping details.

Hospital: Standardizes patient blood pressure readings (e.g., 120/80, 120-80, 120 80 → converted to one format).

### Tools Used:

Python (Pandas, NumPy), Excel (Power Query, Filters), OpenRefine

## Data Analysis (Finding Patterns & Insights)

### What happens?

- Analyze the data using statistical methods and machine learning to discover trends, correlations, and anomalies.

### Real-Life Example

Amazon: Finds that users buy more electronics during festive seasons.

Hospital: Identifies that diabetic patients aged 50+ have a higher risk of heart disease.

### Techniques Used:

Descriptive Analytics → "What happened?" (Sales dropped by 20% last month)

Diagnostic Analytics → "Why did it happen?" (Due to a pricing issue)

Predictive Analytics → "What will happen?" (Forecasts future sales)

Prescriptive Analytics → "What should we do?" (Increase advertising in low-sales regions)

**Tools Used:**

Python (Pandas, NumPy, Scikit-learn), R, Excel (Pivot Tables), SPSS

## **Data Visualization (Making Data Understandable)**

### **What happens?**

- Convert complex data into **charts, graphs, and dashboards** to make it easier to understand.

### **Real-Life Example**

Amazon: Creates a bar chart showing top-selling products by region.

Hospital: Uses a heatmap to show disease spread across different cities.

### **Common Visualizations:**

Bar Chart → Sales by product category

Pie Chart → Market share of different brands

Line Graph → Website traffic trends over time

Heatmap → User activity on a website

**Tools Used:**

Tableau, Power BI, Matplotlib (Python), Excel Charts

## Data-Driven Decision Making (Taking Action)

### What happens?

- Insights are used to make better business decisions, reduce costs, and improve customer satisfaction.

### Real-Life Example

**Amazon:** Uses recommendation algorithms to suggest products based on user history.

**Hospital:** Automates early disease detection using AI-based models.

### Outcome:

Improved efficiency ✓ Cost savings ✓ Better customer experience ✓  
Competitive advantage

## Summary: Data Analytics Process

Step	What Happens?	Real-Life Example	Tools Used
Data Collection	Gather raw data	Amazon collects user purchase data	SQL, APIs, Excel
Data Cleaning	Remove errors, fix missing values	Remove duplicate orders	Python (Pandas), OpenRefine
Data Analysis	Find trends, patterns	Predict sales for next quarter	Python (NumPy), R, Excel
Data Visualization	Create graphs, charts	Show top-selling products	Tableau, Power BI
Decision Making	Take business actions	Recommend products based on user behavior	AI, Machine Learning

## Final Takeaway

**Data Analytics** = Turning raw data into useful insights!

It helps businesses make better decisions, save money, and improve customer experience.

# Excel

## Day 2

### Introduction to Excel & Basic Functions

#### 1. Overview of Excel Interface

Excel is a powerful spreadsheet program that allows users to organize, analyze, and visualize data. The Excel interface includes several key components:

- Ribbon: The ribbon is the top toolbar in Excel that contains tabs (Home, Insert, Page Layout, Formulas, etc.) and groups of related commands. For example, under the "Home" tab, you'll find options for font formatting, alignment, and number formatting.
- Worksheet: A worksheet is a single tab in a workbook, and it consists of rows and columns where data is stored. Each worksheet contains cells arranged in a grid.
- Cells: Cells are individual boxes in the worksheet where data can be entered. Cells are identified by a combination of row and column (e.g., A1, B2).

## 2. Creating, Saving, and Opening Workbooks

- **Creating a New Workbook:** To create a new workbook, you can click on "File" > "New" > "Blank Workbook" or use the shortcut Ctrl + N.
- **Saving a Workbook:** To save a workbook, click "File" > "Save As" and choose the location. You can also use the shortcut Ctrl + S. Excel files are saved with the .xlsx extension by default.
- **Opening an Existing Workbook:** To open an existing workbook, click "File" > "Open" and browse for the file you want to open. You can also use the shortcut Ctrl + O.

## 3. Basic Navigation

- **Cell Referencing:** In Excel, each cell is identified by a unique address consisting of a column letter and row number (e.g., A1 means column A, row 1).
- **Selecting Cells:**
  - Click on a single cell to select it.
  - To select a range of cells, click and drag the mouse over the desired cells.
  - To select an entire row or column, click on the row or column header (e.g., click on the "1" to select row 1, or click on "A" to select column A).

## 4. Basic Formatting

- **Font Styles:** You can change the font style, size, and color using the "Home" tab on the Ribbon. Select the cell(s) and choose your desired font from the font group.
- **Alignment:** You can align the content of cells to the left, center, or right, and also adjust vertical alignment. Use the alignment options on the "Home" tab in the Ribbon.

- Cell Borders: You can add borders to cells to make your data stand out. In the "Home" tab, you'll find the "Borders" option where you can choose from various border styles.
- Colors: You can change the background color of a cell using the "Fill Color" button on the "Home" tab.

## 5. Introduction to Formulas and Functions

Formulas in Excel are used to perform calculations. They always start with an equal sign =.

- **Basic Functions:**
  - **SUM:** Adds a range of numbers.
    - **Syntax:** =SUM(A1:A10)
    - **Example:** If you have numbers in cells A1 through A10, =SUM(A1:A10) will add them all up.
  - **AVERAGE:** Calculates the average of a range of numbers.
    - **Syntax:** =AVERAGE(A1:A10)
    - **Example:** If you have numbers in cells A1 through A10, =AVERAGE(A1:A10) will calculate the average.
  - **COUNT:** Counts the number of cells in a range that contain numbers.
    - **Syntax:** =COUNT(A1:A10)
    - **Example:** =COUNT(A1:A10) will count how many cells from A1 to A10 contain numeric values.
  - **COUNTA:** Counts the number of non-empty cells in a range.
    - **Syntax:** =COUNTA(A1:A10)
    - **Example:** =COUNTA(A1:A10) will count how many cells in the range A1 to A10 are not empty.
  - **MIN:** Finds the minimum value in a range.
    - **Syntax:** =MIN(A1:A10)
    - **Example:** =MIN(A1:A10) will return the smallest value from the range A1 to A10.

- **MAX:** Finds the maximum value in a range.
  - **Syntax:** =MAX(A1:A10)
  - **Example:** =MAX(A1:A10) will return the largest value from the range A1 to A10.

## Cell Referencing in Excel

There are three types of cell referencing in Excel:

1. **Relative Reference:**
  - a. Default referencing style. It changes when you copy the formula to another cell.
  - b. Example: =A1 + B1 in cell C1, when copied to C2, becomes =A2 + B2.
2. **Absolute Reference:**
  - a. Uses dollar signs (\$) to lock the reference to a specific cell.
  - b. Example: =\$A\$1 + \$B\$1. No matter where you copy this formula, it will always refer to cell A1 and B1.
3. **Mixed Reference:**
  - a. Combines relative and absolute references. You can lock either the row or column.
  - b. Example: =A\$1 locks the row (1) but allows the column to change when copied. \$A1 locks the column (A) but allows the row to change when copied.

## Example Walkthrough:

Let's create a simple worksheet where we perform basic functions and use different types of cell referencing:

1. **Step 1: Create a Sample Data Set**
  - a. In cells A1:A5, input the following numbers: 5, 10, 15, 20, 25.
  - b. In cells B1:B5, input the numbers: 3, 6, 9, 12, 15.
2. **Step 2: Apply Basic Functions**

- a. In cell C1, input the formula =SUM(A1:A5). This will return the sum of the numbers in cells A1 through A5.
- b. In cell D1, input the formula =AVERAGE(B1:B5). This will return the average of the numbers in cells B1 through B5.

### **3. Step 3: Apply Relative, Absolute, and Mixed Referencing**

- a. In cell E1, input =A1 \* B1. This is a relative reference. When you copy this formula to cell E2, it will automatically update to =A2 \* B2.
- b. In cell F1, input =\$A\$1 \* B1. This is an absolute reference for A1 but a relative reference for B1. When copied, the reference to A1 will not change, but the reference to B1 will.
- c. In cell G1, input =A\$1 \* B1. This is a mixed reference where the row 1 for A is fixed, but both the column and row for B1 are relative.

### **4. Step 4: Format the Results**

- a. Apply bold and change the font size for the cells that contain formulas.
- b. Use the "Fill Color" option to highlight the cells with results.

By following these steps, you'll have a good understanding of Excel's interface, basic functions, and how to work with different types of cell referencing.

## **Mini Project 1: Sales Data Analysis**

Objective: Analyze sales data and calculate various statistics using Excel's basic functions and formatting.

### **Step-by-Step Implementation:**

#### **Step 1: Create a Workbook and Input Data**

1. Open Excel and create a new workbook by selecting "File" > "New" > "Blank Workbook".
2. In **Sheet1**, input the following sales data for a fictional company:

- a. **Column A:** Product Name (e.g., "Product A", "Product B", "Product C", etc.)
- b. **Column B:** Units Sold
- c. **Column C:** Unit Price
- d. **Column D:** Total Sales (calculated as Units Sold \* Unit Price)

#### **Example data:**

Product Name	Units Sold	Unit Price	Total Sales
Product A	100	10	
Product B	150	15	
Product C	200	20	

#### **Step 2: Calculate Total Sales for Each Product**

1. In **D2**, enter the formula `=B2*C2` to calculate the total sales for the first product.
2. Drag the formula down to the other cells in column D to calculate total sales for all products.

#### **Step 3: Calculate Basic Statistics**

1. **SUM:** In D6, calculate the total sales of all products by entering `=SUM(D2:D4)`.
2. **AVERAGE:** In D7, calculate the average sales by entering `=AVERAGE(D2:D4)`.
3. **COUNT:** In D8, count the number of products by entering `=COUNT(A2:A4)`.
4. **MIN:** In D9, find the product with the least sales by entering `=MIN(D2:D4)`.
5. **MAX:** In D10, find the product with the highest sales by entering `=MAX(D2:D4)`.

#### **Step 4: Format the Worksheet**

1. Apply bold formatting to the headers (Product Name, Units Sold, Unit Price, and Total Sales).
2. Change the background color of the header row to light gray.

3. Adjust the column width to fit the content.
4. Add borders to the cells to make the data visually appealing.

### **Step 5: Save the Workbook**

1. Save the workbook as "Sales\_Data\_Analysis.xlsx" by clicking **File > Save As** and choosing the location.

## **Mini Project 2: Employee Salary Report**

Objective: Create a salary report for employees and apply conditional formatting to highlight high and low salaries.

### **Step-by-Step Implementation:**

#### **Step 1: Create a New Workbook and Input Employee Data**

1. Open Excel and create a new workbook.
2. In Sheet1, input the following employee data:
  - a. Column A: Employee Name (e.g., "John", "Alice", "Bob", etc.)
  - b. Column B: Department
  - c. Column C: Base Salary
  - d. Column D: Bonus
  - e. Column E: Total Salary (calculated as Base Salary + Bonus)

#### **Example data:**

Employee Name	Department	Base Salary	Bonus	Total Salary
John	HR	50000	2000	
Alice	Sales	60000	3000	
Bob	IT	70000	4000	

## Step 2: Calculate Total Salary

1. In **E2**, enter the formula =C2+D2 to calculate the total salary for the first employee.
2. Drag the formula down to other cells in column E to calculate total salaries for all employees.

## Step 3: Apply Basic Functions

1. **SUM:** In **E6**, calculate the total salary of all employees by entering =SUM(E2:E4).
2. **AVERAGE:** In **E7**, calculate the average salary by entering =AVERAGE(E2:E4).
3. **COUNT:** In **E8**, count the number of employees by entering =COUNT(A2:A4).
4. **MIN:** In **E9**, find the employee with the lowest salary by entering =MIN(E2:E4).
5. **MAX:** In **E10**, find the employee with the highest salary by entering =MAX(E2:E4).

## Step 4: Apply Conditional Formatting

1. Select the Total Salary column (E2:E4).
2. Go to the Home tab > Conditional Formatting > Color Scales > choose a color scale (e.g., a green-yellow-red scale). This will highlight high salaries in green and low salaries in red.

## Step 5: Format the Worksheet

1. Apply bold formatting to the header row.
2. Align the text in the center for the "Employee Name" and "Department" columns.
3. Add borders around the cells for a neat appearance.
4. Change the font size of the header row to 12 and make it bold.

## Step 6: Save the Workbook

1. Save the workbook as "Employee\_Salary\_Report.xlsx" by clicking File > Save As and choosing the location.

## Key Learnings from the Mini Projects:

- Creating and Formatting Worksheets: How to create, format, and organize data in Excel.
- Using Basic Functions: How to use Excel functions like SUM, AVERAGE, COUNT, MIN, and MAX.
- Cell Referencing: How to use relative, absolute, and mixed cell referencing.
- Conditional Formatting: How to highlight data using color scales based on specific criteria.
- Saving and Sharing Workbooks: Learn to save workbooks with appropriate names and file types for future use.

These mini projects will help you understand and apply Excel's basic functions, cell referencing, and formatting features, while also giving you practical hands-on experience with real-world data.

## Day 2 tasks:

### 1. Overview of Excel Interface

- Task: Open Excel and identify the Ribbon, Worksheet, and Cells. Explore the Home tab, Insert tab, and Page Layout tab. Write down what functions are available in each tab.

### 2. Creating, Saving, and Opening Workbooks

- Task: Create a new workbook in Excel. Add some data in a few cells. Save the workbook in Excel Workbook (.xlsx) format and then close the file. Reopen the file to verify that your data is intact.

### 3. Basic Navigation - Cell Referencing

- Task: In a new Excel worksheet, input numbers in A1 to A5. Use relative referencing to create a formula in B1 that multiplies the value of A1 by 10. Then drag the formula down to B5 to apply it to the rest of the cells.

#### **4. Basic Navigation - Selecting Cells**

- Task: In a new worksheet, input data in cells A1 to A10. Select different ranges of cells using the Shift key, Ctrl key, and mouse clicks. Highlight a specific column and row together.

#### **5. Basic Formatting - Font Styles**

- Task: In a worksheet, input data in A1 to A5. Change the font of the first cell to bold, the second cell to italic, and the third cell to underline. Apply font color and font size changes as well.

#### **6. Basic Formatting - Cell Alignment**

- Task: Input data in A1 to A3. Align the text to the left, right, and center in each of these cells. Try both horizontal and vertical alignments and note the effects on the data display.

#### **7. Basic Formatting - Cell Borders and Colors**

- Task: Select a range of cells from A1 to C3 and apply a border around the selected cells. Fill the cells with a light blue color. Change the color of the text in one of the cells to white.

#### **8. Introduction to Formulas - SUM**

- Task: In a worksheet, enter numbers in A1 to A5. In A6, use the SUM function to calculate the total of those numbers.

#### **9. Introduction to Formulas - AVERAGE**

- Task: Using the same worksheet, calculate the AVERAGE of the numbers in A1 to A5 using the AVERAGE function in A6.

#### **10. Introduction to Functions - COUNT & COUNTA**

- Task: Enter numbers in cells A1 to A5 and text values in cells B1 to B3. Use the COUNT function in C1 to count the number of numeric entries and the COUNTA function in C2 to count all non-blank entries.

## 11. Introduction to Functions - MIN & MAX

- Task: In a worksheet, input numbers from A1 to A5. Use the MIN function in A6 to find the smallest number, and the MAX function in A7 to find the largest number in the range.

## 12. Cell Referencing - Absolute and Mixed

- Task: In a new worksheet, input values in A1 and B1. In C1, write a formula that multiplies A1 by B1 using absolute cell referencing (e.g., \$A\$1 \* B1). Then in D1, write a formula that uses mixed referencing (e.g., A\$1 \* B1).

## 13. Cell Referencing - Relative Referencing

- Task: Enter data in A1 to A3 and B1 to B3. In C1, create a formula that adds A1 and B1. Drag the formula down to C3 and explain how the formula changes as you copy it to other cells.

## Mini Project 1: Personal Budget Tracker

**Objective:** Create a simple personal budget tracker in Excel where you can track your income and expenses over a month.

### Steps:

1. Overview of Excel Interface:
  - a. Open a new workbook and name it "Personal Budget Tracker."
  - b. Familiarize yourself with the Ribbon, Worksheet, and Cells.
2. Creating, Saving, and Opening Workbooks:
  - a. Create a worksheet with the following columns: Date, Category (e.g., Rent, Groceries, Entertainment), Income, Expense.
  - b. Save the workbook with a suitable name (e.g., "Budget\_2025.xlsx").
3. Basic Navigation:
  - a. Enter data for at least 5 income and expense transactions.
  - b. Use relative references for income and expenses (e.g., formula in the Total Income column).

4. Basic Formatting:

- a. Apply font styles (bold headings), cell borders (for categories and totals), and colors to differentiate sections.
- b. Format cells containing currency as currency (e.g., Income and Expense columns).

5. Using Formulas:

- a. Use the SUM function to calculate total income and total expenses.
- b. Use the AVERAGE function to calculate the average expense per category.
- c. Use the MIN and MAX functions to find the smallest and largest expenses in your tracker.

6. Cell Referencing:

- a. Use absolute referencing to lock the total income cell when calculating individual expenses.

## Mini Project 2: Employee Attendance Tracker

**Objective:** Create an Excel sheet to track the attendance of employees in a month and calculate the total working days.

**Steps:**

1. Overview of Excel Interface:

- a. Open a new workbook and name it "Employee Attendance Tracker."
- b. Explore the Ribbon, Worksheet, and Cells.

2. Creating, Saving, and Opening Workbooks:

- a. Create a worksheet with the following columns: Employee Name, Department, 1st Day, 2nd Day, ..., 30th Day, Total Present Days.
- b. Save the workbook as "Employee\_Attendance\_March\_2025.xlsx."

3. Basic Navigation:

- a. Enter the attendance data for 3 employees, indicating "P" for present, "A" for absent.
  - b. Use relative referencing for the attendance data of each employee.
4. Basic Formatting:
    - a. Apply font styles to headings (bold) and cell borders around the employee's name, attendance data, and total column.
    - b. Use colors to highlight "P" for present (green) and "A" for absent (red).
  5. Using Formulas:
    - a. Use the COUNTIF function to count the total "P" (present) days for each employee.
    - b. Use the MIN and MAX functions to find the employee with the least and most attendance days.
    - c. Use the SUM function to calculate the total present days.
  6. Cell Referencing:
    - a. Use absolute referencing for the attendance data ranges when counting present days.

## Day 3

### Data Formatting & Basic Data Analysis in Excel

This section covers how to format data, apply basic data analysis, and use several essential Excel functions. Let's go step-by-step through each topic.

#### 1. Formatting Numbers, Dates, and Text

##### Definition:

Data formatting allows you to display data in a way that makes it easier to read or understand. In Excel, you can format numbers, dates, and text in various ways.

**Syntax:**

- **Numbers:** Change number formats using the Ribbon or cell formatting options (e.g., currency, percentage, decimal places).
- **Dates:** Format dates to display in different formats (e.g., MM/DD/YYYY or DD-MMM-YY).
- **Text:** Text can be formatted by adjusting alignment, fonts, colors, etc.

**Examples:****1. Formatting Numbers as Currency:**

- a. Select a range of cells with numbers (e.g., sales figures).
- b. Right-click, choose Format Cells, and then select Currency.
- c. The numbers will now display as currency (e.g., \$1,000.00).

**2. Formatting Dates:**

- a. Select a cell with a date (e.g., 01/01/2025).
- b. Right-click, choose Format Cells, then select Date and choose a format like MM/DD/YYYY.
- c. This will display the date as 01/01/2025.

**3. Text Formatting:**

- a. Select the cell with text (e.g., Hello).
- b. Change the font, size, and color from the Ribbon to make it bold or italics.
- c. You can also use Alignment options to center the text.

**2. Using Conditional Formatting (Highlighting Cells Based on Conditions)****Definition:**

Conditional formatting allows you to format cells dynamically based on specific criteria. For example, you can highlight all cells that are greater than a certain number.

**Syntax:**

- Go to the Home tab and select Conditional Formatting.
- Choose a rule, e.g., "Highlight Cell Rules" -> "Greater Than."

**Example:**

**1. Highlighting Cells Greater Than a Value:**

- Select a range of cells with numbers (e.g., sales data).
- Go to the Home tab → Conditional Formatting → Highlight Cell Rules → Greater Than.
- Enter a value (e.g., 500), and Excel will highlight cells greater than 500 with a predefined color.

**2. Color Scales:**

- Select a range of cells.
- Go to Conditional Formatting → Color Scales.
- This will color cells based on their values from lowest to highest (e.g., red for low, green for high).

**3. Introduction to Sorting and Filtering Data**

**Definition:**

Sorting and filtering allow you to view and organize data in a specific order or display only the data that meets certain criteria.

**Syntax:**

- Sort: Use the Sort button on the Ribbon to sort data.
- Filter: Use the Filter option to display only rows that meet specific criteria.

**Examples:**

**1. Sorting Data:**

- Select a column of data (e.g., sales figures).
- Go to the Data tab → Sort.

- c. Choose either Ascending (smallest to largest) or Descending (largest to smallest).

## 2. Filtering Data:

- a. Select the header row of your dataset.
- b. Go to Data → Filter.
- c. Click the filter icon in the column header, and choose the condition to filter the data (e.g., showing only "Sales" greater than 500).

## 4. Using Find & Replace

### Definition:

The Find & Replace tool allows you to search for specific data within your worksheet and replace it with something else.

### Syntax:

- Go to Home → Find & Select → Replace.
- Enter the Find term and Replace term, and Excel will replace the occurrences.

### Example:

#### 1. Replacing Text:

- a. Press Ctrl + H to open the Find & Replace dialog.
- b. In Find what, enter "Apple," and in Replace with, enter "Banana."
- c. Click Replace All to replace all instances of "Apple" with "Banana."

## 5. Introduction to Text Functions

Text functions allow you to manipulate and extract parts of text within cells.

### Functions:

- **CONCATENATE:** Joins multiple strings into one.

- **LEFT:** Extracts a given number of characters from the left side of a string.
- **RIGHT:** Extracts characters from the right side of a string.
- **MID:** Extracts characters from the middle of a string.
- **LEN:** Returns the length of a text string.
- **TRIM:** Removes extra spaces from a string.

**Syntax:**

- **CONCATENATE:** =CONCATENATE(text1, text2, ...)
- **LEFT:** =LEFT(text, num\_chars)
- **RIGHT:** =RIGHT(text, num\_chars)
- **MID:** =MID(text, start\_num, num\_chars)
- **LEN:** =LEN(text)
- **TRIM:** =TRIM(text)

**Examples:****1. CONCATENATE:**

- =CONCATENATE(A1, " ", B1) will combine the text in A1 and B1 with a space in between.

**2. LEFT:**

- =LEFT(A1, 5) will return the first 5 characters from the string in cell A1.

**3. RIGHT:**

- =RIGHT(A1, 3) will return the last 3 characters of the string in cell A1.

**4. MID:**

- =MID(A1, 3, 4) will return 4 characters starting from the 3rd character of the string in A1.

**5. LEN:**

- =LEN(A1) will return the length of the string in A1.

**6. TRIM:**

- =TRIM(A1) will remove any leading, trailing, or extra spaces within the text in A1.

## 6. Working with AutoFill and Flash Fill

### AutoFill:

AutoFill helps you quickly copy or fill in data based on a pattern. For example, you can fill a series of dates or numbers.

#### Syntax:

- Select a cell with data (e.g., 1), drag the fill handle (a small square at the bottom-right corner) down or across to fill cells.

#### Example:

##### 1. AutoFill Series:

- a. Enter 1 in a cell, and 2 in the next.
- b. Select both cells, then drag the fill handle to continue the series (3, 4, 5, etc.).

### Flash Fill:

Flash Fill is used to automatically fill data based on a pattern you provide.

#### Syntax:

- Start typing the pattern in a column (e.g., extracting first names from full names).
- Press Ctrl + E, and Excel will automatically fill the remaining cells based on the pattern.

#### Example:

##### 1. Flash Fill (First Name Extraction):

- a. If you have a full name in A1 like "John Smith," and you type "John" in B1, Excel can automatically extract the first names for all entries in the column by pressing Ctrl + E.

## **Summary:**

These tools and functions in Excel are essential for efficiently working with data, performing basic analysis, and applying formatting. With practice, you'll be able to manipulate and analyze data quickly, making your workflow more efficient.

## **Mini Project 1: Sales Data Analysis**

### **Objective:**

In this mini project, you will analyze sales data by formatting numbers, dates, and text, applying conditional formatting, sorting, and filtering data, and using basic text functions.

### **Step-by-Step Implementation:**

#### **Step 1: Data Preparation**

1. Open a new Excel workbook.
2. Enter the following data in columns A to E:

Order ID	Sales Person	Sale Date	Amount	Region
101	John	01/01/2025	1500	East
102	Jane	03/01/2025	2500	West
103	Mike	05/01/2025	1000	East
104	Sarah	07/01/2025	3500	North
105	John	09/01/2025	1800	South

#### **Step 2: Formatting Numbers, Dates, and Text**

1. **Formatting Sales Amount:**
  - a. Select column D (Amount).
  - b. Right-click and choose Format Cells → Currency. This will display numbers as currency (e.g., \$1,500.00).
2. **Formatting Dates:**

- a. Select column C (Sale Date).
- b. Right-click and choose Format Cells → Date, then choose the format MM/DD/YYYY.

### **3. Formatting Text:**

- a. Select column B (Sales Person).
- b. Use the Home tab to change the font to Bold and Font Size to 12 for better visibility.

## **Step 3: Applying Conditional Formatting**

### **1. Highlight Sales Amount Above \$2,000:**

- a. Select column D (Amount).
- b. Go to Home → Conditional Formatting → Highlight Cell Rules → Greater Than.
- c. Enter 2000 and choose a formatting style (e.g., a green fill with dark green text).

### **2. Highlight Sales for "East" Region:**

- a. Select column E (Region).
- b. Go to Home → Conditional Formatting → Highlight Cell Rules → Text that Contains.
- c. Enter East and choose a different color format.

## **Step 4: Sorting and Filtering**

### **1. Sort Data by Sales Amount (Descending):**

- a. Select the range A1:E6.
- b. Go to Data → Sort → Choose Sales Amount (Column D) → Largest to Smallest.

### **2. Filter Data for Region "East":**

- a. Select the header row (A1:E1).
- b. Go to Data → Filter.
- c. Click the filter icon in column E (Region) and choose "East" to show only sales from the East region.

## Step 5: Using Find & Replace

### 1. Replace "John" with "Johnny":

- Press Ctrl + H to open the Find & Replace dialog.
- In **Find what**, type John, and in **Replace with**, type Johnny.
- Click **Replace All** to change all occurrences.

## Step 6: Using Text Functions

### 1. Concatenate Sales Person and Amount:

- In column F, use the formula =CONCATENATE(B2, " - \$", D2) to combine the sales person's name with their sale amount.

### 2. Extract First Name from Full Name (Assuming Full Name in Column G):

- If the full name is in G2 (e.g., "John Doe"), use the formula =LEFT(G2, FIND(" ", G2)-1) to extract the first name (e.g., "John").

### 3. Trim Extra Spaces:

- In column H, use the formula =TRIM(G2) to remove any extra spaces from the full name in column G.

## Step 7: Final Review and Save

- Review the workbook to ensure all data is properly formatted, sorted, and filtered.
- Save the workbook as SalesDataAnalysis.xlsx.

## Mini Project 2: Employee Data Management

### Objective:

This project will involve managing employee data by formatting text and numbers, using sorting, filtering, and applying text functions to clean and analyze data.

## Step-by-Step Implementation:

### Step 1: Data Preparation

1. Open a new Excel workbook.
2. Enter the following employee data in columns A to D:

Employee ID	Employee Name	Join Date	Salary
E001	Alice Johnson	02/15/2020	55000
E002	Bob Smith	05/23/2018	60000
E003	Charlie Brown	11/12/2021	45000
E004	David Green	03/30/2019	70000
E005	Emma White	07/01/2022	40000

### Step 2: Formatting Numbers, Dates, and Text

1. Format Salary as Currency:
  - a. Select column D (Salary).
  - b. Right-click and choose Format Cells → Currency.
2. Format Join Date:
  - a. Select column C (Join Date).
  - b. Right-click and choose Format Cells → Date, then select the format MM/DD/YYYY.
3. Format Employee Name:
  - a. Select column B (Employee Name).
  - b. Use the Home tab to apply Bold and Center Alignment.

### Step 3: Conditional Formatting

1. Highlight Employees with Salary Above \$55,000:
  - a. Select column D (Salary).
  - b. Go to Home → Conditional Formatting → Highlight Cell Rules → Greater Than.
  - c. Enter 55000 and choose a format like light green fill.
2. Highlight New Joinees (Joined After 2020):
  - a. Select column C (Join Date).

- b. Go to Home → Conditional Formatting → New Rule → Format only cells that contain.
- c. Choose Date Occurring → After and select 01/01/2020.

#### **Step 4: Sorting and Filtering**

##### **1. Sort by Salary (Ascending):**

- a. Select the range A1:D6.
- b. Go to Data → Sort → Choose Salary (Column D) → Smallest to Largest.

##### **2. Filter Employees Who Joined After 2020:**

- a. Select the header row (A1:D1).
- b. Go to Data → Filter.
- c. Click the filter icon in column C (Join Date), select Date Filters, and choose After and enter 01/01/2020.

#### **Step 5: Using Find & Replace**

##### **1. Replace "David Green" with "David Black":**

- a. Press Ctrl + H to open the Find & Replace dialog.
- b. In Find what, type David Green, and in Replace with, type David Black.
- c. Click Replace All.

#### **Step 6: Using Text Functions**

##### **1. Concatenate Employee Name and Join Date:**

- a. In column E, use the formula =CONCATENATE(B2, " - ", C2) to combine the employee's name with their join date.

##### **2. Extract First Name from Employee Name:**

- a. In column F, use the formula =LEFT(B2, FIND(" ", B2)-1) to extract the first name (e.g., "Alice" from "Alice Johnson").

##### **3. Trim Extra Spaces in Employee Name:**

- a. In column G, use the formula =TRIM(B2) to remove any extra spaces in the employee name.

## **Step 7: Final Review and Save**

- Review the worksheet to ensure all formatting, sorting, filtering, and formulas are correct.
- Save the workbook as EmployeeDataManagement.xlsx.

## **Summary:**

Both of these mini projects help you practice various Excel skills like formatting numbers, using conditional formatting, sorting and filtering data, and applying text functions. You will also get hands-on experience with key Excel features such as Find & Replace and AutoFill.

## **Day 3 tasks**

### **Data Formatting & Basic Data Analysis Tasks**

#### **1. Format Numbers as Currency**

- Given a list of sales amounts in column A, format the numbers as currency (e.g., \$1,500.00).

#### **3. Format Dates**

In column B, apply a date format that shows the day, month, and year in the format DD/MM/YYYY.

#### **4. Text Formatting**

Bold the text in the header row and apply center alignment to all the text in the table for better presentation.

#### **5. Apply Conditional Formatting**

In column C (representing sales), highlight any value greater than \$2,000 with a green fill.

## **6. Conditional Formatting - Highlight Top 10%**

Apply conditional formatting to highlight the top 10% of values in column C (Sales) with a specific color.

## **7. Sort Data by Sales Amount**

Given a list of sales and corresponding salespersons, sort the data by sales amount in descending order.

## **8. Filter Data by Date**

Filter the list of data to show only sales records that occurred after 01/01/2022.

## **9. Find & Replace Data**

Use the **Find & Replace** feature to replace all instances of the word "Expired" with "Out of Stock" in a given list of products.

## **10.Extract First Name from Full Name**

In column D, extract the first name from a list of full names in column A using a text function (e.g., "John Doe" → "John").

## **11.Concatenate First and Last Name**

In column E, use the CONCATENATE function to combine the first name in column A and last name in column B, separated by a space.

## **12.Calculate the Length of Text in a Cell**

In column F, calculate the length of the text in column A using the **LEN** function. For example, "Hello World" → 11.

## **13.Remove Extra Spaces from Text**

In column G, use the **TRIM** function to remove any leading or trailing spaces from the text in column A.

#### **14. Use AutoFill to Fill a Series**

Create a series of numbers in column H starting from 1 and use **AutoFill** to extend it to 100.

### **Mini Project 1: Monthly Expense Tracker**

#### **Objective:**

Create a monthly expense tracker where you will format numbers, use conditional formatting, and perform basic analysis to manage personal finances.

#### **Steps:**

##### **1. Data Entry:**

- a. Create columns for:
  - i. Expense Category (e.g., Groceries, Rent, Entertainment)
  - ii. Date of Expense
  - iii. Amount Spent
- b. Example data:
  - i. Groceries | 01/03/2022 | \$150
  - ii. Rent | 01/03/2022 | \$1200
  - iii. Entertainment | 03/03/2022 | \$200

##### **2. Formatting:**

- a. Format the Amount Spent column as currency.
- b. Format the Date of Expense column to show the date in DD/MM/YYYY format.

##### **3. Conditional Formatting:**

- a. Highlight all expenses over \$500 in red.

**4. Sorting & Filtering:**

- a. Sort the data by Expense Category alphabetically.
- b. Filter the data to show only expenses made in March 2022.

**5. Basic Data Analysis:**

- a. Use SUM to calculate the total amount spent for the month.
- b. Use AVERAGE to calculate the average spending amount.
- c. Use COUNT to count how many expenses were logged.

## **Mini Project 2: Student Grades Analysis**

**Objective:**

Create a student grades analysis sheet where you will format grades, calculate averages, and apply text functions to extract student information.

**Steps:**

**1. Data Entry:**

- a. Create columns for:
  - i. Student Name
  - ii. Exam Date
  - iii. Grade
- b. Example data:
  - i. John Doe | 15/03/2022 | 85
  - ii. Jane Smith | 16/03/2022 | 92
  - iii. Mark Johnson | 17/03/2022 | 76

**2. Text Formatting:**

- a. Use LEFT and RIGHT functions to extract and separate first and last names into separate columns.

**3. Date Formatting:**

- a. Format the Exam Date column to display the date in DD/MM/YYYY format.

**4. Using Conditional Formatting:**

- Highlight grades above 90 with green and grades below 70 with red.

**5. Text Functions:**

- Use LEN to calculate the length of each student's name and display it in a new column.

**6. Basic Data Analysis:**

- Use AVERAGE to calculate the average grade across all students.
- Use MIN and MAX to identify the highest and lowest grades in the list.
- Use COUNTIF to count how many students scored above 80.

**7. Sorting:**

- Sort the data by Grade in descending order to rank students from highest to lowest.

## Day 4

### Advanced Formulas and Functions in Excel

#### 1. Introduction to IF Statements

An IF statement is used to test a condition and returns one value if the condition is true and another value if the condition is false. This helps to make decisions in Excel based on the criteria provided.

##### Syntax:

IF(logical\_test, value\_if\_true, value\_if\_false)

- logical\_test: The condition that you want to test.
- value\_if\_true: The value to return if the condition is true.
- value\_if\_false: The value to return if the condition is false.

**Example:**

Let's say you want to assign "Pass" or "Fail" based on a student's score (score > 50 is a pass).

=IF(A1 > 50, "Pass", "Fail")

- If A1 is greater than 50, the result will be "Pass".
- If A1 is less than or equal to 50, the result will be "Fail".

## **2. IF, AND, OR, IFERROR**

### **AND Function**

The AND function checks if all conditions are true and returns TRUE only if all the conditions are met.

#### **Syntax:**

AND(logical1, logical2, ...)

#### **Example (using IF and AND):**

You can use AND within IF to check multiple conditions. For example, checking if a student's score is greater than 50 and if the attendance is above 75%.

=IF(AND(A1 > 50, B1 > 75), "Pass", "Fail")

- A1 is the score, B1 is the attendance. The result will be "Pass" if both conditions are true, otherwise "Fail".

## OR Function

The OR function checks if any of the conditions are true. It returns TRUE if at least one condition is met.

### Syntax:

`OR(logical1, logical2, ...)`

### Example (using IF and OR):

You can use OR with IF to check if either condition is true. For example, checking if a student either passed the exam or had good attendance.

`=IF(OR(A1 > 50, B1 > 75), "Pass", "Fail")`

- This formula returns "Pass" if either condition is true.

## IFERROR Function

The IFERROR function is used to catch and handle errors in formulas, like dividing by zero or looking up a value that doesn't exist.

### Syntax:

`IFERROR(value, value_if_error)`

### Example:

`=IFERROR(A1/B1, "Error")`

- If dividing A1 by B1 results in an error (e.g., dividing by zero), it will return "Error". Otherwise, it will return the result of the division.

### 3. Lookup Functions: VLOOKUP and HLOOKUP

#### VLOOKUP (Vertical Lookup)

The VLOOKUP function searches for a value in the first column of a table or range and returns a value in the same row from another column.

##### Syntax:

`VLOOKUP(lookup_value, table_array, col_index_num, [range_lookup])`

- `lookup_value`: The value to search for.
- `table_array`: The range of cells containing the data.
- `col_index_num`: The column number from which to return the value.
- `[range_lookup]`: Optional. TRUE for approximate match (default), or FALSE for exact match.

##### Example:

`=VLOOKUP(A2, B1:D5, 3, FALSE)`

- This searches for the value in A2, looks in the range B1:D5, and returns the value from the 3rd column (C).

#### HLOOKUP (Horizontal Lookup)

The HLOOKUP function is similar to VLOOKUP, but it searches for a value in the first row of a table and returns a value from the same column in another row.

##### Syntax:

`HLOOKUP(lookup_value, table_array, row_index_num, [range_lookup])`

##### Example:

`=HLOOKUP(A2, B1:F3, 2, FALSE)`

- This searches for the value in A2, looks in the range B1:F3, and returns the value from the 2nd row.

## 4. Introduction to INDEX & MATCH (Alternative to VLOOKUP)

### INDEX Function

The INDEX function returns the value of a cell within a range based on its row and column number.

#### Syntax:

`INDEX(array, row_num, [column_num])`

### MATCH Function

The MATCH function returns the position of a value within a range.

#### Syntax:

`MATCH(lookup_value, lookup_array, [match_type])`

### Example (INDEX & MATCH combined):

If you want to find the corresponding value in the second column for a matching value in the first column:

`=INDEX(B2:B5, MATCH("Product2", A2:A5, 0))`

- MATCH("Product2", A2:A5, 0) finds the row number of "Product2" in column A.
- INDEX(B2:B5, ...) returns the value from column B in the same row.

## 5. Using CHOOSE and INDIRECT for Dynamic Referencing

### CHOOSE Function

The CHOOSE function returns a value based on the index number you provide.

#### Syntax:

CHOOSE(index\_num, value1, value2, ...)

#### Example:

=CHOOSE(2, "Red", "Green", "Blue")

- This will return "Green" because the index is 2.

### INDIRECT Function

The INDIRECT function returns the reference specified by a text string.

#### Syntax:

INDIRECT(ref\_text)

#### Example:

=INDIRECT("A1")

- This will return the value in cell A1.

## 6. Text Functions: UPPER, LOWER, PROPER, TEXT

### UPPER Function

The UPPER function converts all letters in a text string to uppercase.

**Syntax:**

UPPER(text)

**Example:**

=UPPER("hello")

- The result is "HELLO".

**LOWER Function**

The LOWER function converts all letters in a text string to lowercase.

**Syntax:**

LOWER(text)

**Example:**

=LOWER("HELLO")

- The result is "hello".

**PROPER Function**

The PROPER function capitalizes the first letter of each word in a text string.

**Syntax:**

PROPER(text)

**Example:**

=PROPER("hello world")

- The result is "Hello World".

## TEXT Function

The TEXT function formats a number or date as text in a specific format.

### Syntax:

`TEXT(value, format_text)`

### Example:

`=TEXT(TODAY(), "mm/dd/yyyy")`

- This formats the current date as "mm/dd/yyyy".

## Mini Project 1: Sales Report Analysis using Advanced Formulas

### Project Overview:

In this project, you will create a sales report to analyze product sales performance using IF Statements, AND/OR, VLOOKUP, INDEX/MATCH, and TEXT functions. The project will involve analyzing sales data, checking for specific conditions like sales targets, and dynamically referencing values to create a comprehensive report.

### Step-by-Step Implementation:

#### Step 1: Prepare the Sales Data Table

1. Create a new Excel sheet and name it "Sales Report".
2. In the sheet, create the following columns:
  - a. Product ID (Column A)
  - b. Product Name (Column B)
  - c. Sales Amount (Column C)
  - d. Target Sales (Column D)
  - e. Sales Status (Column E)

**Sample Data:**

Product ID	Product Name	Sales Amount	Target Sales	Sales Status
101	Product A	550	500	
102	Product B	450	500	
103	Product C	600	600	

**Step 2: Apply Conditional Logic to Determine Sales Status**

Use **IF** statements to check if the product has met its sales target.

- Formula for the "Sales Status" column (Column E):

=IF(C2 >= D2, "Target Achieved", "Target Not Achieved")

- This formula checks if the Sales Amount (C2) is greater than or equal to the Target Sales (D2). If the condition is true, it returns "Target Achieved", otherwise "Target Not Achieved".

**Step 3: Use AND/OR Functions to Combine Multiple Conditions**

In some cases, you might want to evaluate multiple conditions. For example, checking if the sales amount is greater than a specific value **AND** the product's sales status has been "Target Achieved".

- Formula to check multiple conditions:

=IF(AND(C2 >= 500, E2 = "Target Achieved"), "Bonus Eligible", "No Bonus")

- This formula checks if the Sales Amount is greater than or equal to 500 **AND** the Sales Status is "Target Achieved". If both conditions are true, it returns "Bonus Eligible".

#### **Step 4: Use VLOOKUP to Retrieve Product Information**

Now, create another table with product information and use VLOOKUP to find the corresponding Product Name and display it based on the Product ID.

- Additional Data Table: | Product ID | Product Name | Category | |-----|-----| | 101 | Product A | Electronics | | 102 | Product B | Furniture | | 103 | Product C | Apparel |
- Formula to look up product name based on Product ID:

=VLOOKUP(A2, \$A\$8:\$C\$10, 2, FALSE)

- This formula searches for the Product ID in the range A8:C10 and returns the Product Name from the second column.

#### **Step 5: Use INDEX & MATCH for Dynamic Lookup**

To replace VLOOKUP, you can use INDEX & MATCH functions, which offer more flexibility.

- Formula to retrieve Product Name using INDEX & MATCH:

=INDEX(\$B\$8:\$B\$10, MATCH(A2, \$A\$8:\$A\$10, 0))

- MATCH finds the position of the Product ID in the range A8:A10, and INDEX returns the corresponding Product Name from column B.

#### **Step 6: Use Text Functions for Formatting and Concatenation**

You can use UPPER, LOWER, PROPER, and TEXT functions to format the text data.

- Formula to capitalize the first letter of the product name:

=PROPER(B2)

- This converts "product a" into "Product A".
- Formula to format the sales amount as currency:

=TEXT(C2, "\$#,##0.00")

- This converts 550 into "\$550.00".

### **Step 7: Final Touches**

Apply Conditional Formatting to highlight products that are "Bonus Eligible" or "Target Achieved".

- Go to Home > Conditional Formatting and choose a format for your criteria (e.g., highlight cells containing "Bonus Eligible" in green).

## **Mini Project 2: Employee Performance Report using Advanced Functions**

### **Project Overview:**

In this project, you will build an employee performance report using IF Statements, CHOOSE, INDIRECT, VLOOKUP, AND/OR, and TEXT functions. The project will track employee performance and assign them a category based on their performance ratings.

### **Step-by-Step Implementation:**

#### **Step 1: Prepare the Employee Data Table**

Create a new sheet and name it "Employee Performance". Add the following columns:

- Employee ID (Column A)
- Employee Name (Column B)

- Performance Score (Column C)
- Category (Column D)
- Salary Increase (Column E)

### Sample Data:

Employee ID	Employee Name	Performance Score	Category	Salary Increase
001	John Doe	85		
002	Jane Smith	90		
003	Sam Brown	70		

### Step 2: Use IF Statements for Performance Category

Use an IF statement to assign a category based on the performance score.

- **Formula** for the "Category" column:

=IF(C2 >= 90, "Excellent", IF(C2 >= 75, "Good", "Needs Improvement"))

- This formula assigns "Excellent" if the score is 90 or above, "Good" if the score is between 75 and 89, and "Needs Improvement" if the score is below 75.

### Step 3: Use CHOOSE and INDIRECT for Dynamic Reference

You can use the CHOOSE function for assigning specific categories.

- **Formula** using **CHOOSE**:

=CHOOSE(MATCH(C2, {0,75,90}, 1), "Needs Improvement", "Good", "Excellent")

- This formula uses **MATCH** to find where the score fits within the range and then uses **CHOOSE** to return the corresponding category.

### Step 4: Use VLOOKUP to Retrieve Salary Information

Assume there is a table with salary information based on categories. Use VLOOKUP to look up the salary increase based on the employee's category.

- Additional Data Table: | Category | Salary Increase | |-----|-----  
-----| | Excellent | 15% | | Good | 10% | | Needs Improvement | 5% |
- Formula for Salary Increase:

=VLOOKUP(D2, \$G\$2:\$H\$4, 2, FALSE)

- This looks up the Category in the range G2:H4 and returns the corresponding salary increase.

### Step 5: Use Text Functions for Formatting

You can use TEXT to format the salary increase as a percentage.

- **Formula** to format salary increase:

=TEXT(E2, "0%")

- This will display 15% as "15%" in the salary increase column.

### Step 6: Use AND/OR for Multiple Conditions

You might want to check multiple conditions to give employees a bonus.

- **Formula** to check multiple conditions:

=IF(AND(C2 >= 85, D2 = "Excellent"), "Bonus Eligible", "No Bonus")

- This formula checks if the performance score is 85 or higher **AND** if the category is "Excellent". If both conditions are true, it returns "Bonus Eligible", otherwise "No Bonus".

## Conclusion:

These two mini projects will help you practice using advanced Excel formulas and functions to analyze data. By combining multiple functions like IF, VLOOKUP, INDEX/MATCH, TEXT, and logical functions like AND/OR, you'll be able to create dynamic, efficient reports and make complex calculations in Excel.

## Day 4 tasks

### 1. IF Statements

- Create a formula that checks if the sales in a given month exceed a target value. If the sales exceed the target, return "Target Met," otherwise return "Target Not Met."

### 2. Nested IF Statements

- Use a nested IF formula to assign grades based on scores. If the score is above 90, assign "A"; between 80 and 90, assign "B"; between 70 and 80, assign "C"; and below 70, assign "F."

### 3. AND/OR Functions

- Use AND and OR to check if an employee meets the conditions for a promotion. The employee must have a performance rating of 80 or above AND must have worked for more than 5 years. If either of the conditions is false, the result should be "Not Eligible."

### 4. IFERROR Function

- Create a formula that checks if a division by zero occurs when dividing the total sales by the number of employees. If an error occurs, return "Error," otherwise return the result of the division.

## 5. VLOOKUP - Exact Match

- Use VLOOKUP to find the price of a product based on its product ID. Assume the product ID is in column A and the product prices are in a separate table in columns C and D, where column C contains product IDs and column D contains prices. Use exact match for the lookup.

## 6. VLOOKUP - Approximate Match

- Use VLOOKUP to find the correct tax rate for an income range. The tax rate table contains income ranges in column A and corresponding tax rates in column B. Perform an approximate match lookup.

## 7. HLOOKUP

- Create a formula using HLOOKUP to find the price of a product based on the product's name. The product names are listed horizontally in row 1, and their corresponding prices are in row 2.

## 8. INDEX & MATCH (Alternative to VLOOKUP)

- Use INDEX and MATCH to find the product name based on a given product ID. The product IDs are in column A, and the product names are in column B. MATCH will find the row number, and INDEX will return the product name.

## 9. INDEX & MATCH with Multiple Criteria

- Create a formula using INDEX and MATCH to find the salary of an employee based on both their department and job title. The employee data is in a table with columns for Department, Job Title, and Salary.

## 10. CHOOSE Function

- Use the CHOOSE function to return the sales category ("High", "Medium", or "Low") based on the sales value. If the sales are greater than 1000, return "High"; between 500 and 1000, return "Medium"; and below 500, return "Low."

## 11. INDIRECT Function

- Use the INDIRECT function to reference a cell address dynamically. For example, if cell A1 contains the text "B5", use INDIRECT to reference the value in cell B5.

## 12. Text Functions - CONCATENATE

- Use the CONCATENATE function (or &) to combine a first name in column A and a last name in column B into a full name in column C.

## 13. Text Functions - UPPER, LOWER, PROPER

- Use the UPPER, LOWER, and PROPER functions to format a list of names. Convert all names to uppercase, lowercase, and proper case (first letter of each name capitalized) respectively.

## Mini Project 1: Student Performance and Grading System

**Objective:** Create a system to track student performance, calculate their grades, and determine whether they pass or fail based on multiple criteria.

### Requirements:

#### 1. Data Setup:

- a. Columns: Student ID, Student Name, Math Score, Science Score, English Score, Total Marks, Grade, Pass/Fail.
2. **IF Statement for Grade Calculation:**
  - a. Use an IF statement to assign grades based on the Total Marks (out of 300):
    - i. If Total Marks  $\geq$  270, assign grade "A".
    - ii. If Total Marks  $\geq$  240, assign grade "B".
    - iii. If Total Marks  $\geq$  210, assign grade "C".
    - iv. If Total Marks  $<$  210, assign grade "D".
3. **AND/OR Function for Pass/Fail Determination:**
  - a. Use AND to determine if a student has passed. A student passes if their score in Math, Science, and English is greater than or equal to 50.
  - b. If all three conditions are met, return "Pass"; otherwise, return "Fail."
4. **VLOOKUP for Student Category:**
  - a. Use VLOOKUP to retrieve the category (e.g., "Regular", "Distinguished") for each student from a lookup table that maps Student ID to their category.
5. **Text Functions for Formatting:**
  - a. Use UPPER to display the Student Name in uppercase in a new column.
  - b. Use PROPER to display the Student Name in proper case in another column.
6. **Conditional Formatting:**
  - a. Apply conditional formatting to the Grade column. Highlight "A" in green, "B" in yellow, and "C" and "D" in red.

## **Mini Project 2: Product Inventory Management System**

**Objective:** Create an inventory management system that tracks product details and stock levels.

**Requirements:**

- 1. Data Setup:**
  - a. Columns: Product ID, Product Name, Stock Level, Reorder Level, Reorder Quantity, Supplier Name.
- 2. IFERROR for Stock Calculation:**
  - a. Use IFERROR to avoid errors when calculating the total value of products in stock. If there is an error in the calculation (e.g., non-numeric data), return "Invalid Input."
- 3. VLOOKUP for Supplier Information:**
  - a. Use VLOOKUP to find the supplier's contact details based on the Product ID from a lookup table.
- 4. INDEX & MATCH for Reorder Level Search:**
  - a. Use INDEX and MATCH to search for products where the Stock Level is below the Reorder Level, and return the Product Name and Stock Level.
- 5. CHOOSE for Stock Status:**
  - a. Use CHOOSE to create a new column for the Stock Status:
    - i. If Stock Level is greater than or equal to the Reorder Level, return "In Stock."
    - ii. If Stock Level is below Reorder Level, return "Reorder."
- 6. INDIRECT for Dynamic Cell Reference:**
  - a. Use INDIRECT to reference a cell dynamically. For example, if a user inputs the column reference in cell A1 (like "B"), use INDIRECT to reference column B for product stock.
- 7. Text Functions:**
  - a. Use TEXT to format the Stock Level as currency (e.g., "\$1,000").
  - b. Use CONCATENATE to combine Product Name and Supplier Name into a single column.

# Day 5

## Working with Tables and Data Validation in Excel

### 1. Introduction to Excel Tables and Benefits

Excel tables are structured ranges of data with specific features and benefits, such as sorting, filtering, and easily referencing data.

**Definition:** An Excel Table is a data range that has been formatted with specific properties for better organization and management.

- **Benefits of Excel Tables:**

- Automatic range expansion: When new data is entered, the table automatically expands to include it.
- Sorting and Filtering: Tables allow you to sort and filter data easily with just a click.
- Structured References: You can use column names directly in formulas rather than cell references (e.g., =SUM(Table1[Sales]) instead of =SUM(B2:B10)).

#### How to Create an Excel Table:

1. Select the range of data you want to convert into a table.
2. Go to the **Insert** tab and click on **Table**.
3. Make sure the "My table has headers" checkbox is selected if your data includes headers.
4. Click **OK**.

**Example:** Let's say you have a list of sales data, including columns for Product, Quantity Sold, and Sales Amount. You can convert this range into a table and easily sort by Sales Amount or filter by Product Type.

## 2. Data Validation (Restricting Data Entry)

Data validation in Excel helps you restrict what users can enter into a cell, ensuring data consistency and preventing errors.

**Definition:** Data Validation allows you to set rules on the type of data that can be entered into a particular cell or range.

- **Types of Data Validation:**

- Whole Number: Restrict data to integers.
- Decimal: Allow decimal numbers.
- List: Create a drop-down list of allowed entries.
- Date/Time: Restrict data to specific date or time ranges.

### How to Apply Data Validation:

1. Select the cell or range of cells where you want to apply validation.
2. Go to the Data tab and click on Data Validation.
3. In the Settings tab, select the type of data validation you want to apply.
4. Configure the specific validation criteria (e.g., list, number range, etc.).
5. Optionally, you can create input messages and error alerts for users.

### Example - Creating Drop-down Lists:

1. Select a range of cells where you want to create a drop-down list (e.g., column Product Type).
2. Go to Data → Data Validation → List.
3. In the Source box, enter the items you want in the drop-down (e.g., Electronics, Clothing, Furniture).
4. Click **OK**. Now, only these options can be selected from the drop-down in each cell.

### Example - Restricting Data to Whole Numbers:

1. Select the cell or range where only whole numbers should be entered.

2. Go to Data → Data Validation → Whole Number.
3. Set a range, for example, between 1 and 100 to restrict users to only whole numbers within that range.

### 3. Using Named Ranges

Named Ranges make it easier to refer to a specific range of cells in formulas, making them more readable and easy to manage.

**Definition:** A Named Range is a descriptive name assigned to a range of cells or a single cell in Excel.

#### How to Create a Named Range:

1. Select the range of cells you want to name (e.g., A1:A10).
2. Go to the Formulas tab and click Define Name.
3. Enter a name for the range (e.g., SalesData).
4. Click OK.

**Example:** If you have a named range **SalesData** for the range A1:A10, you can reference it in a formula like this:

=SUM(SalesData)

### 4. Subtotal & Grouping

Subtotal and Grouping are used to analyze data and organize large datasets by summarizing data into different levels.

- **Subtotal:** Adds a summary for each group of data (e.g., sum, average).
- **Grouping:** Collapses data into groups for better viewing.

### How to Use Subtotal:

1. Sort the data by the column you want to group (e.g., by Product Type).
2. Go to the Data tab and click on Subtotal.
3. Choose the column to group by and the function to use (e.g., sum).
4. Excel will add subtotals for each group of data.

### How to Use Grouping:

1. Select a range of rows.
2. Go to the Data tab and click Group.
3. Excel will group the rows, and you can collapse/expand them using the +/- buttons.

## 5. Introduction to Data Consolidation

**Data Consolidation** allows you to combine data from different ranges, sheets, or workbooks into a single table.

**Definition:** Data Consolidation is used when you have data in multiple locations and want to consolidate it into one place for analysis.

### How to Use Data Consolidation:

1. Go to the Data tab and click Consolidate.
2. Choose the function you want to use (e.g., Sum, Average).
3. Select the ranges to consolidate and click Add.
4. You can choose whether to link the data or not.
5. Click OK to consolidate the data.

**Example:** If you have monthly sales data in separate worksheets for January, February, and March, you can use data consolidation to combine all data into a single summary sheet.

## Summary

Here's a quick summary of the concepts discussed:

- Excel Tables: Structured data with sorting, filtering, and dynamic range expansion.
- Data Validation: Restricting data entry with criteria like numbers, dates, or drop-down lists.
- Named Ranges: Assigning descriptive names to data ranges for easier reference in formulas.
- Subtotal & Grouping: Summarizing and organizing data for better analysis.
- Data Consolidation: Merging data from multiple sources into one place.

## Mini Project 1: Sales Data Analysis with Excel Tables and Data Validation

### Project Overview:

In this project, we will create a sales data analysis sheet where you manage and analyze sales data. The goal is to utilize Excel Tables, Data Validation, Named Ranges, and Subtotal/Grouping to make the data entry and analysis process more organized and efficient.

### Step-by-Step Implementation:

#### Step 1: Create and Organize Data with Excel Table

##### 1. Create Sales Data:

- a. Open a new Excel workbook.
- b. Add columns such as Date, Product Name, Quantity Sold, Sales Amount, Salesperson.
- c. Enter sample data for these columns.

**Example data:**

Date	Product Name	Quantity Sold	Sales Amount	Salesperson
01/01/2025	Laptop	5	5000	Alice
01/01/2025	Mouse	10	2000	Bob
02/01/2025	Laptop	3	3000	Alice
02/01/2025	Keyboard	7	1400	Bob

**2. Convert Data to Table:**

- Select the data range.
- Go to the Insert tab → Table.
- Make sure My table has headers is selected and click OK.

**Step 2: Apply Data Validation for Salesperson Column****1. Create Drop-down List for Salesperson:**

- Select the Salesperson column.
- Go to the Data tab → Data Validation.
- In the Data Validation window, choose List from the drop-down.
- In the Source box, type: Alice, Bob, Charlie, Dave.
- Click OK. Now, only these values can be selected in the Salesperson column.

**Step 3: Apply Named Ranges for Columns****1. Name the Sales Amount Column:**

- Select the Sales Amount column (excluding the header).
- Go to the Formulas tab → Define Name.
- Name the range SalesData.
- Now, you can refer to this range in formulas by using SalesData instead of cell references.

**2. Use Named Ranges in Formulas:**

- In a new cell, type the formula to calculate the total sales:

=SUM(SalesData)

#### **Step 4: Group and Subtotal Data by Salesperson**

1. **Sort Data by Salesperson:**
  - a. Select the entire table.
  - b. Go to the Data tab → Sort.
  - c. Choose Salesperson as the column to sort by.
2. **Apply Subtotal:**
  - a. With the data sorted by Salesperson, go to the Data tab → Subtotal.
  - b. In the Subtotal dialog box, select Sales Amount as the column to sum, and choose Salesperson as the column to group by.
  - c. Click OK. Excel will add subtotals for each salesperson.

#### **Step 5: Data Consolidation**

1. **Consolidate Monthly Sales:**
  - a. If you have sales data in multiple sheets (e.g., January, February, March), go to the sheet where you want to consolidate data.
  - b. Go to Data tab → Consolidate.
  - c. Select Sum as the function.
  - d. Click Add and select the ranges from the different sheets.
  - e. Click OK to consolidate the data into one sheet.

### **Mini Project 2: Inventory Management with Data Validation and Grouping**

#### **Project Overview:**

This project focuses on building an inventory management system with features such as drop-down lists for item types, data validation for stock levels, and grouping for items based on categories.

## Step-by-Step Implementation:

### Step 1: Create Inventory Data

#### 1. Create Inventory List:

- Open a new Excel workbook.
- Create columns like Item ID, Item Name, Category, Stock Level, Unit Price.

#### Example data:

Item ID	Item Name	Category	Stock Level	Unit Price
101	Laptop	Electronics	50	1000
102	Mouse	Electronics	200	25
201	T-shirt	Clothing	150	20
202	Jeans	Clothing	80	40

#### 2. Convert Data to Table:

- Select the range of your data and go to **Insert → Table**.
- Make sure the **My table has headers** box is checked and click **OK**.

### Step 2: Apply Data Validation for Category Column

#### 1. Create Drop-down List for Categories:

- Select the Category column.
- Go to Data → Data Validation → List.
- In the Source box, type: Electronics, Clothing, Furniture, Accessories.
- Click OK. This restricts users to selecting a valid category from the list.

### Step 3: Data Validation for Stock Level

#### 1. Restrict Stock Level to Positive Numbers:

- Select the Stock Level column.
- Go to Data → Data Validation → Whole Number.
- In the Data box, choose greater than or equal to and set the minimum value to 0.

- d. Click OK. This ensures that users can only enter non-negative numbers.

#### **Step 4: Named Ranges for Price and Stock Levels**

1. Name the Stock Level and Unit Price Columns:
  - a. Select the Stock Level column (excluding the header).
  - b. Go to Formulas → Define Name, and name the range StockLevel.
  - c. Similarly, define Price for the Unit Price column.
2. Use Named Ranges in Formulas:
  - a. In a new cell, use the named range to calculate the total inventory value:

=SUMPRODUCT(StockLevel, Price)

#### **Step 5: Grouping Items by Category**

1. Sort Data by Category:
  - a. Select the entire table.
  - b. Go to the Data tab → Sort.
  - c. Choose Category as the column to sort by.
2. Apply Grouping:
  - a. After sorting by category, select the rows you want to group (e.g., all items in the Electronics category).
  - b. Go to Data → Group.
  - c. Excel will add a plus/minus button to collapse and expand the grouped data.

#### **Step 6: Data Consolidation**

1. Consolidate Inventory Data from Multiple Sources:
  - a. If you have inventory data in separate sheets for different stores (e.g., Store 1, Store 2), go to the sheet where you want to consolidate the data.
  - b. Go to Data → Consolidate.

- c. Select Sum as the function and click Add to select ranges from different sheets.
- d. Click OK to consolidate the data into one sheet.

## **Summary:**

In both projects, we demonstrated how to:

- Work with Excel Tables to organize data, sort, and filter.
- Apply Data Validation to restrict data entry with drop-down lists and custom rules.
- Use Named Ranges to simplify referencing data in formulas.
- Group and Subtotal data to make analysis more efficient.
- Perform Data Consolidation to combine data from multiple sources into one.

## **Real-Life Mini Project: Employee Performance Management System**

### **Project Overview:**

In this project, you will create an **Employee Performance Management System** using Excel. The goal is to track employee performance over several months, manage and analyze performance data, and ensure that the data entry is accurate and organized.

### **Day 5 Tasks:**

1. Create Employee Performance Data:
  - a. Create an Excel worksheet with the following columns: Employee ID, Employee Name, Department, Month, Performance Score, Salary.
  - b. Populate the worksheet with sample data for at least 10 employees over 6 months.

2. Convert Data into an Excel Table:
  - a. Convert the data into an Excel Table by selecting the data range and choosing Insert > Table.
  - b. Ensure the My table has headers option is checked.
3. Apply Sorting and Filtering to the Table:
  - a. Sort the data by Employee Name alphabetically.
  - b. Filter the data to show only employees from a specific department (e.g., Sales or Marketing).
4. Create a Drop-down List for Departments:
  - a. Select the Department column and apply Data Validation to create a drop-down list with options such as Sales, Marketing, HR, IT, etc.
  - b. This will restrict data entry to these valid department names.
5. Apply Data Validation for Performance Score:
  - a. Apply Data Validation to the Performance Score column to restrict data entry to whole numbers between 1 and 10.
  - b. This ensures only valid performance scores are entered.
6. Use Named Ranges for Employee IDs and Performance Scores:
  - a. Name the Employee ID range as EmployeeID and the Performance Score range as PerformanceScore.
  - b. Use these named ranges in formulas to simplify references.
7. Calculate Average Performance Score for Each Employee:
  - a. Use the AVERAGE function to calculate the average performance score for each employee across all the months they have worked.
  - b. Use the named range PerformanceScore to reference the data.
8. Group Data by Department:
  - a. Group the employee data by Department so that each department's performance can be analyzed separately.
  - b. Use the Group feature in Excel (Data > Group) to collapse or expand each department's data.
9. Apply Subtotals to Calculate Average Performance by Department:

- a. After sorting the data by Department, use the Subtotal feature to calculate the average performance score for each department.
- b. Select Performance Score as the column to average and group by Department.

10. Create a Dynamic Summary Table for Employee Performance:

- a. Create a separate sheet where you summarize the performance of each department and individual employee using Pivot Tables.
- b. Include the Employee Name, Department, and Average Performance Score in the pivot table.

11. Use Conditional Formatting to Highlight Top Performers:

- a. Apply Conditional Formatting to the Performance Score column to highlight employees with scores above 8 in green, and those with scores below 5 in red.
- b. This will help visually distinguish high and low performers.

12. Consolidate Data from Multiple Sheets (e.g., Different Locations):

- a. If you have multiple sheets with employee performance data for different locations, use the Consolidate feature to combine all the data into one sheet.
- b. Choose the Sum or Average function to combine the data across locations.

13. Create a Data Dashboard for Employee Performance:

- a. Use Excel's charting tools to create a dashboard that displays key metrics like average performance scores, top-performing employees, and departmental performance over time.
- b. This should be done using Charts, Pivot Tables, and Slicers for interactivity.

**Summary:**

This mini project will help you understand and apply various Excel features such as Tables, Data Validation, Named Ranges, Grouping, Subtotals, Consolidation, and Conditional Formatting in a real-world context. You will also learn how to

create summary reports and dashboards using Pivot Tables and Charts, which will be helpful for managing and analyzing employee performance data in a structured and dynamic way.

## **Mini Project 1: Employee Attendance Tracker**

### **Project Overview:**

Create an Employee Attendance Tracker in Excel to monitor employee attendance, holidays, and performance. You will use tables, data validation, and conditional formatting to generate reports for HR.

### **Day 5 Tasks:**

1. Create an Employee Attendance Table: Design an Excel table with columns such as Employee ID, Employee Name, Date, Status (Present/Absent), Reason for Absence, and Holiday.
2. Apply Data Validation for Status: Use data validation to create a drop-down list for the Status column that allows only "Present" or "Absent" as valid options.
3. Use Named Ranges: Create named ranges for Employee Name and Status columns to make referencing easier in formulas.
4. Apply Sorting and Filtering: Sort the attendance data by Employee Name and Date. Use filtering to show records for specific employees or dates.
5. Conditional Formatting for Absenteeism: Highlight Absent status entries in red and Present status entries in green using conditional formatting.
6. Group Attendance Data by Month: Group the data by month using Excel's Group feature to analyze monthly attendance patterns.
7. Create a Summary Report by Employee: Use Pivot Tables to create a summary report showing total present and absent days for each employee.

8. Data Validation for Reason for Absence: Use data validation to restrict entries in the Reason for Absence column to a predefined list, such as "Sick," "Vacation," "Personal," etc.
9. Track Holidays: Add a Holiday column where a drop-down list allows the HR department to mark holidays.
10. Apply Subtotal Feature: Use the Subtotal feature to calculate the total number of days an employee was absent or present in a given month.
11. Visualize Attendance Data: Create a bar chart showing the number of Absent and Present days for each employee.
12. Create Conditional Formatting for Repeated Absence: Highlight employees with more than 5 absences in a month in yellow using conditional formatting.
13. Prepare an Attendance Dashboard: Build a dashboard with key metrics, such as total absentees, average attendance rate, and employee performance using Pivot Charts and slicers.

## Mini Project 2: Student Grade Tracker

### Project Overview:

Create a Student Grade Tracker in Excel to track student performance, calculate final grades, and analyze the results. You will use Excel tables, data validation, and conditional formatting to ensure accurate data entry and analysis.

### Day 5 Tasks:

1. Create a Grade Tracker Table: Design a table with columns such as Student ID, Student Name, Subject, Assignment 1, Assignment 2, Midterm, Final Exam, and Final Grade.
2. Apply Data Validation for Grades: Use data validation to restrict entries in Assignment, Midterm, and Final Exam columns to numbers between 0 and 100.

3. Use Named Ranges: Create named ranges for the Assignment, Midterm, and Final Exam columns for easier reference in formulas.
4. Sort and Filter by Subject: Sort the data by Student Name and filter by Subject to view specific subjects' grades.
5. Calculate Final Grade: Use the IF and SUM functions to calculate the Final Grade by giving specific weight to assignments and exams.
6. Conditional Formatting for Low Grades: Apply conditional formatting to highlight grades below 50% in red.
7. Group Data by Subject: Group data by Subject to analyze average grades and performance for each subject.
8. Create a Drop-down List for Subjects: Use data validation to create a drop-down list for the Subject column to select from a list of predefined subjects (e.g., Math, Science, English).
9. Apply Subtotals to Calculate Average Grades: Use the Subtotal feature to calculate the average grade for each subject.
10. Track Student Performance Trends: Create a line chart to track students' performance over the semester for different assignments and exams.
11. Data Validation for Final Grades: Restrict the Final Grade column to accept only grades between 0 and 100.
12. Highlight Students Below Passing Grade: Use conditional formatting to highlight students who scored below the passing grade (e.g., 60%) in yellow.
13. Prepare a Grade Summary Report: Create a report summarizing the students' performance, including average grades, top performers, and students who need improvement, using Pivot Tables and Pivot Charts.

# Day 6

## Working with Pivot Tables

A Pivot Table is a powerful tool in Excel that allows you to summarize, analyze, explore, and present large sets of data. It enables you to aggregate and manipulate data in various ways, making it easier to extract insights.

### 1. Introduction to Pivot Tables

A Pivot Table in Excel helps you automatically summarize large amounts of data in an interactive table format. You can use Pivot Tables to group, filter, and perform calculations on your data.

### 2. Creating and Modifying Pivot Tables

#### Step-by-Step Process:

1. Prepare Your Data:
  - a. Ensure your data is in tabular form with headers in the first row (e.g., Employee ID, Name, Department, Salary).
  - b. The data should be clean and free of empty rows or columns.
2. Creating a Pivot Table:
  - a. Select your data range.
  - b. Go to the Insert tab on the Ribbon and click PivotTable.
  - c. In the Create PivotTable dialog box, ensure the correct data range is selected.
  - d. Choose where to place the PivotTable (New Worksheet or Existing Worksheet).

**Example:** Imagine you have a dataset with employee names, departments, and salaries. You want to create a PivotTable to analyze total salaries by department.

**Data:**

<b>Employee ID</b>	<b>Name</b>	<b>Department</b>	<b>Salary</b>
1	John	HR	50000
2	Jane	IT	60000
3	Mary	HR	55000
4	Mark	IT	70000
5	Lucy	Finance	80000

**Steps:**

- e. Select the entire dataset.
  - f. Click Insert → PivotTable.
  - g. Select New Worksheet and click OK.
3. Designing the Pivot Table:
- a. In the PivotTable Field List on the right:
    - i. Drag fields like Department to the Rows area.
    - ii. Drag the Salary field to the Values area.
    - iii. The PivotTable will now show the total salary for each department.

**3. Grouping Data, Adding Subtotals**

**Grouping** data in a PivotTable allows you to summarize and analyze your data at different levels. You can group data based on dates, numbers, or categories.

**Example: Grouping Dates by Month:**

If you have a dataset with sales data including sale dates, you can group them by month and year.

**Data Example:**

Sale Date	Amount
01/15/2023	200
02/10/2023	300
03/05/2023	400

**Steps to Group by Month:**

1. Insert a PivotTable from your data.
2. Drag Sale Date to the Rows field.
3. Right-click on any date in the PivotTable, and choose Group.
4. In the Group By dialog, select Months and Years.
5. This will create a summary showing sales per month.

**Adding Subtotals:**

To add subtotals, simply drag a field to both the **Rows** and **Values** areas. Excel will automatically calculate subtotals for each group.

## 4. Using Pivot Charts

Pivot Charts are graphical representations of data in a PivotTable. They help to visualize trends and patterns in your data.

**Steps to Create a Pivot Chart:**

1. Click anywhere inside the PivotTable.
2. Go to the PivotTable Analyze tab and click PivotChart.
3. Select the chart type (e.g., column chart, pie chart) and click OK.
4. Excel will create a PivotChart linked to your PivotTable.

**Example:**

Using the employee salary data from earlier, a Bar Chart can show the total salary by department. This helps to visually compare the salary distribution across different departments.

## 5. Calculated Fields in Pivot Tables

Calculated Fields allow you to create custom formulas within a PivotTable to perform calculations on data fields.

### **Example of a Calculated Field:**

You have employee data, and you want to calculate the Bonus as 10% of the Salary.

1. Click anywhere inside the PivotTable.
2. Go to the PivotTable Analyze tab and click Fields, Items & Sets → Calculated Field.
3. In the Insert Calculated Field dialog box, name your calculated field (e.g., Bonus).
4. In the Formula box, type =Salary\*0.1 (assuming the field is named "Salary").
5. Click OK, and a new field will be added to your PivotTable showing the bonus for each department or employee.

## 6. Filtering Pivot Tables by Date and Value

PivotTables can be filtered to focus on specific data. You can filter by date, values, or even specific criteria using the Filter field.

### **Filtering by Date:**

1. Drag a Date field to the Rows area of the PivotTable.
2. Click on the drop-down arrow next to any date field in the PivotTable and select Date Filters.
3. Choose filters like "This Month," "Next Year," or custom date ranges.

### **Filtering by Values:**

1. Drag a field like Sales to the Values area.
2. Click the drop-down arrow in the Values field and choose Value Filters.
3. For example, filter to show only Sales > 500.

## Real Life Example Recap:

**Scenario:** Imagine you're tracking employee salaries across various departments in a company. Using a PivotTable, you can:

- Summarize the total salaries per department.
- Group employees by department and visualize salary trends using Pivot Charts.
- Calculate bonuses for each department using calculated fields.
- Use date filters to focus on data from a specific month or quarter.

## Summary:

- Pivot Tables are essential for summarizing, analyzing, and exploring large data sets.
- Grouping data allows you to see patterns at different aggregation levels (e.g., month, year).
- Pivot Charts offer a visual representation of the data, making it easier to understand.
- Calculated Fields allow custom formulas for on-the-fly calculations inside the PivotTable.
- Filters help you focus on the most relevant data, such as sales in a specific range or products in a specific category.

## Mini Project 1: Sales Data Analysis and Report

### Overview:

This project involves analyzing sales data for a retail store. You will create a Pivot Table to summarize sales performance by product category, salesperson, and

date. Additionally, you will visualize this data using Pivot Charts and calculate commissions for each salesperson using calculated fields.

### **Steps for Mini Project 1:**

#### **Step 1: Prepare the Data**

Create a dataset with the following columns:

- Salesperson: The name of the salesperson.
- Product Category: The category of the product sold.
- Sale Date: The date of the sale.
- Sales Amount: The amount of the sale.

#### **Sample Data:**

Salesperson	Product Category	Sale Date	Sales Amount
John	Electronics	01/15/2023	500
Jane	Furniture	02/10/2023	300
John	Electronics	02/20/2023	700
Mark	Clothing	01/15/2023	200
Lucy	Furniture	03/05/2023	600
Jane	Electronics	03/12/2023	150

#### **Step 2: Insert a Pivot Table**

1. Select the dataset (including headers).
2. Go to the Insert tab and click on PivotTable.
3. In the Create PivotTable dialog box, choose to place the Pivot Table in a New Worksheet.
4. Click OK.

#### **Step 3: Design the Pivot Table**

1. Drag the Product Category field to the Rows area to group the data by product category.

2. Drag the Sales Amount field to the Values area to calculate the total sales per product category.
3. Drag the Salesperson field to the Columns area to analyze sales by salesperson.
4. The Pivot Table should now show total sales by product category and salesperson.

#### **Step 4: Group the Data by Month**

1. Drag the Sale Date field to the Rows area below Product Category.
2. Right-click on any date in the Pivot Table and select Group.
3. In the Group By dialog box, select Months and Years to group the sales by month and year.

#### **Step 5: Add Subtotals**

1. Right-click on any Product Category in the Pivot Table.
2. Click Subtotal and choose Automatic. Excel will automatically add subtotals for each product category.

#### **Step 6: Create a Pivot Chart**

1. Click anywhere inside the Pivot Table.
2. Go to the PivotTable Analyze tab and click PivotChart.
3. Choose a Column Chart or any other chart type to visualize the sales by product category and salesperson.
4. Click OK, and Excel will generate a Pivot Chart based on the Pivot Table data.

#### **Step 7: Add a Calculated Field for Commission**

1. Click anywhere inside the Pivot Table.
2. Go to the PivotTable Analyze tab, click Fields, Items & Sets, and select Calculated Field.
3. Name the calculated field Commission and enter the formula:  
=Sales Amount \* 0.05

4. Click OK. A new column will be added to the Pivot Table, showing the commission (5% of the sales amount) for each salesperson.

### **Step 8: Filter the Pivot Table by Date**

1. Click the drop-down arrow in the Sale Date field in the Pivot Table.
2. Select Date Filters → This Month or any other date filter to view sales for specific time periods.

## **Mini Project 2: Employee Salary and Department Analysis**

### **Overview:**

This project involves analyzing employee salary data across various departments. You will use Pivot Tables to summarize the total salary by department, calculate average salaries, and filter data based on specific conditions.

### **Steps for Mini Project 2:**

#### **Step 1: Prepare the Data**

Create a dataset with the following columns:

- Employee Name: The name of the employee.
- Department: The department to which the employee belongs.
- Salary: The salary of the employee.
- Hire Date: The date the employee was hired.

#### **Sample Data:**

Employee Name	Department	Salary	Hire Date
John	HR	50000	01/15/2020
Jane	IT	60000	02/10/2019
Mark	Marketing	55000	03/05/2021
Lucy	IT	70000	04/15/2020
Mary	HR	45000	06/05/2018

Mark	Marketing	75000	08/12/2021
------	-----------	-------	------------

### Step 2: Insert a Pivot Table

1. Select the dataset (including headers).
2. Go to the Insert tab and click PivotTable.
3. In the Create PivotTable dialog box, choose to place the Pivot Table in a New Worksheet.
4. Click OK.

### Step 3: Design the Pivot Table

1. Drag the Department field to the Rows area to group the data by department.
2. Drag the Salary field to the Values area. The default calculation will be Sum. This will show the total salary per department.
3. Drag the Salary field again to the Values area and change the calculation to Average by right-clicking on the field and selecting Value Field Settings → Average.
4. The Pivot Table should now show both the total and average salary per department.

### Step 4: Group the Data by Year

1. Drag the Hire Date field to the Rows area below Department.
2. Right-click on any hire date in the Pivot Table and select Group.
3. In the Group By dialog box, select Years to group the data by the year employees were hired.

### Step 5: Add Subtotals for Each Department

1. Right-click on any Department field in the Pivot Table.
2. Select Subtotal → Automatic. This will add subtotals for each department.

### Step 6: Filter the Pivot Table by Department

1. Click the drop-down arrow in the Department field in the Pivot Table.
2. Select specific departments (e.g., IT or HR) to filter the data and analyze only the chosen departments.

### **Step 7: Create a Pivot Chart**

1. Click anywhere inside the Pivot Table.
2. Go to the PivotTable Analyze tab and click PivotChart.
3. Choose a Bar Chart to visualize the total and average salaries by department.
4. Click OK, and Excel will generate a Pivot Chart based on the Pivot Table data.

### **Step 8: Calculate the Total Salary for Each Department Using a Calculated Field**

1. Click anywhere inside the Pivot Table.
2. Go to the PivotTable Analyze tab, click Fields, Items & Sets, and select Calculated Field.
3. Name the calculated field Salary Increase and enter the formula:  
=Salary \* 0.10
4. Click OK, and a new column will appear, showing the salary increase for each employee.

### **Step 9: Filter by Specific Salary Range**

1. Click the drop-down arrow in the Salary field.
2. Select Value Filters → Greater Than or Between to filter employees based on their salary.

## **Conclusion:**

Both projects allow you to explore Pivot Tables, Pivot Charts, and calculated fields while focusing on real-life data analysis tasks:

- Mini Project 1: Focuses on sales analysis, including grouping data by date, adding subtotals, and calculating commissions.
- Mini Project 2: Focuses on employee salary analysis, including grouping by department, calculating total and average salaries, and filtering by hire date or salary range.

## Mini Project: Employee Performance and Salary Analysis

### Overview:

This project will involve analyzing employee performance and salary data. The goal is to use Pivot Tables to summarize the total salary, average performance scores, and other key metrics for employees across different departments. You'll also be visualizing the data using Pivot Charts and applying filters for better data analysis.

### Day 6 Tasks):

1. Prepare the Data
2. Create a dataset with the following columns:
  - a. Employee Name
  - b. Department
  - c. Performance Score
  - d. Salary
  - e. Hire Date
3. Insert a Pivot Table

Select the dataset and create a Pivot Table to analyze employee salary and performance data. Place the Pivot Table in a new worksheet.

4. Group Data by Department

In the Pivot Table, drag the Department field to the Rows area to group the data by department. This will allow you to see the breakdown of salary and performance score per department.

5. Add Salary and Performance Data to the Values Area

Drag the Salary field to the Values area and set it to display the sum of salaries for each department. Similarly, drag the Performance Score field to the Values area and set it to display the average performance score per department.

#### 6. Apply Subtotals for Each Department

Right-click any department in the Pivot Table and apply subtotals to show the total salary and average performance score for each department.

#### 7. Group Employees by Hire Date (Year)

Drag the Hire Date field to the Rows area and group the employees by year of hire. This will allow you to see the distribution of employees hired each year within each department.

#### 8. Add a Pivot Chart

Create a Pivot Chart from the Pivot Table to visualize the total salary and average performance score by department. Choose a column chart or bar chart to clearly display the differences across departments.

#### 9. Apply Date Filters

Apply a filter to the Hire Date field to show only employees hired within a specific year range (e.g., employees hired after 2015).

#### 10. Calculate Total Salary Per Department Using a Calculated Field

Create a calculated field to calculate the total salary by adding a formula to the Pivot Table. For example, calculate a new field for the Salary Increase using the formula: =Salary \* 0.10.

#### 11. Filter Data by Performance Score

Apply a filter on the Performance Score field to show only departments where the average performance score is above a certain threshold (e.g., 75%).

#### 12.Add a Calculated Field for Bonus Calculation

Add a calculated field to the Pivot Table to calculate the employee bonus. Assume that employees with a performance score above 80 get a 10% bonus, and those below 80 get a 5% bonus. Use an IF formula to calculate this bonus.

#### 13.Use the Value Filters to Show Salaries Above a Specific Amount

Apply a Value Filter to the Salary field to only show departments where the total salary is greater than \$100,000.

#### 14.Create a Dynamic Pivot Table Using Grouping by Month

Change the grouping in the Hire Date field to group by month instead of year. This will give a more detailed view of the employee hiring trend over time.

### **End Goal of the Project:**

- Summarize the total salary and average performance score for each department.
- Use filters to analyze employees based on their hiring date, performance scores, and salary.
- Visualize the data using Pivot Charts.
- Perform advanced calculations such as salary increases and bonuses using calculated fields.

By completing this mini project, you will gain practical experience with creating and modifying Pivot Tables, adding calculated fields, and using filters and Pivot Charts to analyze real-life data.

## Mini Project 1: Sales Performance Analysis

### Project Requirements:

1. Data Preparation:
  - a. Create a dataset that contains sales data with the following columns:
    - i. Salesperson Name
    - ii. Region
    - iii. Product
    - iv. Sales Amount
    - v. Sales Date
2. Insert a Pivot Table:
  - a. Create a Pivot Table from the sales dataset and place it in a new worksheet.
3. Group Sales Data by Region and Product:
  - a. Group the data in the Pivot Table by Region and Product to analyze sales performance by product and region.
4. Summarize Total Sales:
  - a. Add the Sales Amount field to the Values area to calculate the total sales amount for each combination of Region and Product.
5. Apply Subtotals:
  - a. Apply subtotals to show the total sales per Region.
6. Use Pivot Charts for Visualization:
  - a. Create a Pivot Chart to visualize sales performance across regions and products, using a suitable chart type like a bar chart.
7. Filter Sales Data by Date:
  - a. Use the Sales Date field to filter the data, and group the sales data by quarters or years to analyze trends over time.
8. Add a Calculated Field:
  - a. Add a calculated field to compute the Commission for each sale. Assume that the commission rate is 5% of the sales amount.
9. Apply Value Filters:

- a. Use a Value Filter to show only regions or products where the total sales amount is greater than a certain threshold (e.g., \$50,000).

10. Analyze Sales Performance Over Time:

- a. Apply a Date filter to only display data from the last year and analyze the sales performance for that period.

## **Mini Project 2: Employee Compensation and Benefits Analysis**

### **Project Requirements:**

1. Data Preparation:
  - a. Create a dataset containing the following columns for employees:
    - i. Employee Name
    - ii. Department
    - iii. Salary
    - iv. Bonus
    - v. Hire Date
2. Insert a Pivot Table:
  - a. Create a Pivot Table to analyze employee salary and bonus data by department.
3. Group Employee Data by Department:
  - a. Group the employees by their Department in the Pivot Table to calculate the total salary and bonus for each department.
4. Calculate the Total Salary and Bonus:
  - a. Add the Salary and Bonus fields to the Values area to display the total salary and bonus per department.
5. Add Subtotals for Departments:
  - a. Apply subtotals to show the total compensation (salary + bonus) for each department.
6. Use Pivot Charts for Visualization:
  - a. Create a Pivot Chart to visualize the total salary and bonus by department, using a suitable chart type such as a pie or bar chart.

7. Filter Employees by Hire Date:
  - a. Apply a filter to show employees who were hired in the last 5 years and analyze their compensation data.
8. Add a Calculated Field for Total Compensation:
  - a. Add a calculated field to calculate the total compensation by summing Salary and Bonus.
9. Use Value Filters for Salary Data:
  - a. Apply a Value Filter to show only departments where the total salary is greater than a certain threshold (e.g., \$200,000).
10. Group Employees by Year of Hire:
  - a. Group employees by year of hire using the Hire Date field, and display the total compensation for employees hired each year.

## Day 7

### Charts and Data Visualization in Excel

#### 1. Introduction to Charts in Excel

Charts are graphical representations of data used to visualize patterns, trends, and relationships within a dataset. Excel provides various types of charts, including Column, Line, Pie, and Bar charts, to help interpret data effectively.

#### 2. Common Chart Types in Excel

Chart Type	Purpose
Column Chart	Best for comparing different categories of data.
Line Chart	Ideal for showing trends over time.
Pie Chart	Used to display proportions of a whole.
Bar Chart	Similar to a column chart but displayed horizontally.

## Creating and Customizing Charts in Excel

### Syntax & Step-by-Step Implementation

Let's assume we have sales data for different products:

Product	Sales (Jan)	Sales (Feb)	Sales (Mar)
Laptop	5000	6000	5500
Mobile	7000	7500	7800
Tablet	3000	3200	3100
Headphones	4000	4200	4500

#### Step 1: Insert a Column Chart

1. Select the data range:
  - a. Highlight A1:D5 (Product and sales data).
2. Go to the "Insert" tab in the Ribbon.
3. Select "Column Chart" from the "Charts" group.
4. Choose a chart type, such as a Clustered Column Chart.
5. The chart appears on the worksheet.

#### Step 2: Customize the Chart

- Add Chart Title:
  - Click on the chart → "Chart Elements" (+ sign) → Select "Chart Title" → Type "Monthly Sales Comparison".
- Add Data Labels:
  - Click on the chart → "Chart Elements" (+ sign) → Select "Data Labels".
- Change Axis Titles:
  - Click on "Chart Elements" → Select "Axis Titles".
  - Edit X-axis as "Products" and Y-axis as "Sales Amount".

### 3. Working with Line Charts

A Line Chart is useful for showing trends over time.

#### Step-by-Step Implementation

1. Select Data: Highlight A1:D5.
2. Go to "Insert" tab → Click on "Line Chart".
3. Choose "Line with Markers".
4. Customize the Chart:
  - a. Add a title: "Sales Trend Over Months".
  - b. Format the lines using different colors.

#### Real-Life Example

A company tracks its revenue for 6 months:

Month	Revenue (\$)
Jan	50,000
Feb	55,000
Mar	53,000
Apr	60,000
May	62,000
Jun	58,000

By plotting a **Line Chart**, they can see whether revenue is increasing or decreasing.

### 4. Using Pie Charts

A Pie Chart is useful for displaying percentage distributions.

#### Step-by-Step Implementation

1. Select the Data: Assume you have a dataset for market share:

Company	Market Share (%)
Apple	35%
Samsung	30%
OnePlus	15%
Google	10%
Others	10%

2. Go to "Insert" Tab → Click "Pie Chart".
3. Choose "3D Pie" for better visualization.
4. Customize:
  - a. Add percentages by clicking on "Chart Elements" → "Data Labels".
  - b. Change colors for better clarity.

### Real-Life Example

A company wants to visualize how much revenue each department contributes. Instead of using numbers, they use a Pie Chart to show percentages.

## 5. Using Combo Charts

A Combo Chart is a combination of two chart types, such as a Column and Line Chart together.

### Step-by-Step Implementation

1. Select the data:

Month	Sales (\$)	Profit (\$)
Jan	50000	5000
Feb	55000	5500
Mar	53000	5200
Apr	60000	6000

2. Go to "Insert" Tab → Click "Combo Chart".
3. Choose "Custom Combo Chart".
4. Set "Sales" as Column Chart and "Profit" as Line Chart.

## 5. Customize:

- a. Use different colors for Sales and Profit.
- b. Add a second axis for better readability.

### Real-Life Example

A business owner wants to compare monthly sales (bars) and profit (line) on the same chart.

## 6. Using Sparklines for Trend Visualization

Sparklines are small in-cell charts that show trends.

### Step-by-Step Implementation

#### 1. Select the Data Range:

- a. Example dataset:

Product	Jan	Feb	Mar	Trend
Laptop	5000	6000	5500	
Mobile	7000	7500	7800	
Tablet	3000	3200	3100	

- 2. Go to "Insert" Tab → Click "Sparklines".
- 3. Choose "Line Sparklines".
- 4. Select the Trend column (E2:E4) and click OK.
- 5. The Sparklines appear in column E.

### Real-Life Example

Retail stores use Sparklines in inventory sheets to track product demand.

## 7. Conditional Formatting with Icons

Conditional Formatting allows you to highlight data based on conditions.

## Step-by-Step Implementation

1. Select the Data Range (e.g., sales figures).
2. Go to "Home" Tab → Click "Conditional Formatting".
3. Choose "Icon Sets" → Select Traffic Lights.
4. Set Conditions:
  - a. Green: Sales > 7000
  - b. Yellow: Sales between 4000-7000
  - c. Red: Sales < 4000

## Real-Life Example

A company wants to monitor performance ratings:

- Green = Exceeds expectations
- Yellow = Meets expectations
- Red = Needs improvement

## Conclusion

Charts and data visualization in Excel help in making data-driven decisions. By using various charts, combo charts, Sparklines, and conditional formatting, businesses can analyze trends, track performance, and present insights effectively.

## Mini Project 1: Sales Performance Dashboard

### Project Overview:

A retail company wants to analyze the monthly sales performance of different product categories and visualize the data effectively. You will create an Excel Sales Dashboard that includes Column, Line, Pie, and Bar Charts, Combo Charts, Sparklines, and Conditional Formatting.

**Step-by-Step Implementation:****Step 1: Prepare Sales Data**

Create the following dataset in an Excel sheet:

Month	Electronics (\$)	Clothing (\$)	Groceries (\$)	Total Sales (\$)
Jan	5000	3000	2000	10000
Feb	6000	3500	2500	12000
Mar	7000	4000	3000	14000
Apr	8000	4500	3500	16000
May	8500	4700	3700	16900

**Step 2: Insert a Column Chart**

1. Select data: Highlight A1:D6.
2. Go to "Insert" → Select "Column Chart" → Choose "Clustered Column".
3. Customize the Chart:
  - a. Click on "Chart Elements" (+ sign).
  - b. Add Chart Title: "Monthly Sales by Category".
  - c. Enable Data Labels to show sales figures.

**Step 3: Add a Line Chart for Total Sales**

1. Select "Total Sales (\$)" column.
2. Go to "Insert" → Select "Line Chart".
3. Modify the Line Chart:
  - a. Change the line color for better visibility.
  - b. Add markers at data points.

**Step 4: Create a Pie Chart for Sales Contribution**

1. Select the total sales for each category in May (B6:D6).
2. Go to "Insert" → Choose "Pie Chart".
3. Customize the Pie Chart:
  - a. Add percentages to slices.

- b. Add a legend.

#### **Step 5: Insert a Combo Chart (Column + Line)**

1. Select A1:E6 (including Total Sales).
2. Go to "Insert" → Click "Combo Chart".
3. Set "Electronics, Clothing, Groceries" as Column Chart.
4. Set "Total Sales" as Line Chart.
5. Enable "Secondary Axis" for Total Sales.

#### **Step 6: Use Sparklines for Quick Trend Analysis**

1. Select E2:E6 (Total Sales column).
2. Go to "Insert" → Choose "Sparklines" → "Line".
3. Set the location range in column F.
4. Click OK → The trend appears inside cells.

#### **Step 7: Apply Conditional Formatting with Icons**

1. Select Total Sales (\$) column.
2. Go to "Home" → Click "Conditional Formatting".
3. Choose "Icon Sets" → "Traffic Lights".
4. Set conditions:
  - a. Green: Sales > 15,000
  - b. Yellow: Sales between 12,000 and 15,000
  - c. Red: Sales < 12,000

#### **Final Output:**

- A Sales Dashboard with interactive charts.
- Data visualizations that help in decision-making.
- Conditional formatting highlights important insights.

## Mini Project 2: Employee Attendance & Productivity Report

### Project Overview:

An HR department wants to analyze employee attendance and productivity trends using Excel charts and data visualization tools. You will create an Employee Attendance & Productivity Report that includes Bar Charts, Line Charts, Combo Charts, Sparklines, and Conditional Formatting with Icons.

### Step-by-Step Implementation:

#### Step 1: Prepare Employee Attendance Data

Create the following dataset in an Excel sheet:

Month	Employee	Working Days	Present Days	Absent Days	Productivity (%)
Jan	John	22	20	2	91%
Feb	John	20	18	2	85%
Mar	John	23	21	2	92%
Apr	John	21	19	2	90%
May	John	22	21	1	95%

#### Step 2: Create a Bar Chart for Attendance Analysis

1. Select data: Highlight C1:E6.
2. Go to "Insert" → Select "Bar Chart" → Choose "Stacked Bar".
3. Customize the Chart:
  - a. Add Chart Title: "Employee Attendance Report".
  - b. Enable Data Labels to show the exact attendance figures.
  - c. Change the color for Present Days (Green) and Absent Days (Red).

### **Step 3: Add a Line Chart for Productivity Trend**

1. Select "Month" and "Productivity (%)" columns.
2. Go to "Insert" → Select "Line Chart".
3. Modify the Line Chart:
  - a. Change the line color to Blue for better visibility.
  - b. Add markers to highlight each month's productivity percentage.

### **Step 4: Insert a Combo Chart (Bar + Line)**

1. Select A1:F6 (including Productivity %).
2. Go to "Insert" → Click "Combo Chart".
3. Set "Present & Absent Days" as Bar Chart.
4. Set "Productivity (%)" as Line Chart.
5. Enable "Secondary Axis" for Productivity %.

### **Step 5: Use Sparklines for Quick Trend Analysis**

1. Select F2:F6 (Productivity % column).
2. Go to "Insert" → Choose "Sparklines" → "Line".
3. Set the location range in column G.
4. Click OK → The trend appears inside cells.

### **Step 6: Apply Conditional Formatting with Icons**

1. Select Productivity (%) column.
2. Go to "Home" → Click "Conditional Formatting".
3. Choose "Icon Sets" → "Arrows" (Green, Yellow, Red).
4. Set conditions:
  - a. Green Arrow: Productivity above 90%.
  - b. Yellow Arrow: Productivity between 85% and 90%.
  - c. Red Arrow: Productivity below 85%.

### **Final Output:**

- A professional Employee Attendance & Productivity Report.
- Easy-to-understand visual trends using charts.
- Conditional formatting to highlight key insights.

## Mini Project: Sales Performance & Revenue Analysis Dashboard

### Project Overview:

A company wants to analyze its monthly sales and revenue performance using Excel Charts and Data Visualization. This mini-project will help create a Sales Performance Dashboard to track key business insights using Column, Line, Pie, and Bar Charts, Combo Charts, Sparklines, and Conditional Formatting with Icons.

### Day 7 Tasks

#### 1. Create the Sales Dataset

Prepare a dataset with the following columns:

- Month
- Region
- Total Sales (Units Sold)
- Revenue (in \$)
- Profit (%)

#### 2. Insert a Column Chart for Monthly Sales

- Select Month and Total Sales columns.
- Go to Insert → Column Chart.
- Add Data Labels and Title.

#### 3. Create a Line Chart for Revenue Trends

- Select Month and Revenue columns.
- Insert a Line Chart.
- Customize with Markers and Labels.

#### 4. Use a Pie Chart for Regional Sales Distribution

- Select Region and Total Sales.
- Insert a Pie Chart.

- Customize with Legend and Data Labels.

## 5. Apply a Bar Chart for Top 5 Best-Selling Months

- Sort the Total Sales column in descending order.
- Select the top 5 months and insert a Bar Chart.
- Customize chart colors.

## 6. Create a Combo Chart for Sales & Profit Comparison

- Select Month, Sales, and Profit (%).
- Insert a Combo Chart (Sales as Bar, Profit as Line).
- Enable Secondary Axis for Profit.

## 7. Add Sparklines for Revenue Trends

- Select Revenue Column.
- Insert Sparklines (Line) for quick visualization.

## 8. Apply Conditional Formatting to Highlight High/Low Sales

- Use Color Scales to highlight months with high and low sales.
- Green for highest sales, Red for lowest sales.

## 9. Apply Conditional Formatting Icons for Profitability

- Use Green/Yellow/Red Icons based on profit percentage.

## 10. Add a Dynamic Drop-down Filter for Region Selection

- Use Data Validation to create a drop-down list for filtering by region.

## 11. Use Data Bars for Revenue Column

- Apply Conditional Formatting → Data Bars to visually compare revenues.

## 12. Group Data by Quarter and Analyze Sales Trends

- Use Grouping to consolidate months into Q1, Q2, etc.
- Analyze trends using Pivot Tables.

## 13. Finalize the Dashboard with Titles, Legends, and Formatting

- Add titles, legends, and formatting for a professional dashboard look.

**Expected Outcome:**

- A clear and visually appealing Sales Performance Dashboard.
- Ability to analyze trends, filter data, and track sales performance.
- Use of various charts, sparklines, and conditional formatting for insights.

**Mini Project 1: Employee Performance Dashboard**

**Project Requirement:**

A company wants to track employee performance over the past year based on key metrics such as monthly targets achieved, efficiency percentage, and ratings. The dashboard should include:

- Column Chart for monthly performance trends.
- Line Chart for efficiency percentage over time.
- Pie Chart for department-wise performance distribution.
- Combo Chart (Targets vs. Achieved vs. Efficiency %).
- Sparklines to show performance trends in each department.
- Conditional Formatting Icons to highlight employees exceeding or underperforming their targets.

**Mini Project 2: Customer Order Analysis Report**

**Project Requirement:**

An e-commerce company wants to analyze monthly customer orders and revenue trends to identify key sales patterns. The report should include:

- Bar Chart to compare total orders per month.
- Line Chart to track monthly revenue growth.
- Pie Chart showing product category sales distribution.
- Combo Chart (Orders, Revenue, and Profit % comparison).

- Sparklines to show order trends per product category.
- Conditional Formatting Icons to highlight high/low revenue months.

## Day 8

### Advanced Data Analysis in Excel: A Detailed Guide

Advanced data analysis in Excel allows users to predict trends, optimize decisions, and analyze relationships between variables using various built-in tools. Let's go through these concepts step by step, including definitions, syntax, and real-life examples.

#### 1. Using What-If Analysis

##### What is What-If Analysis?

What-If Analysis allows users to explore **different scenarios** by changing input values and observing the impact on output. Excel provides three What-If Analysis tools:

1. Goal Seek – Find the required input value to achieve a specific output.
2. Data Table – Analyze multiple input values and their impact on an output formula.
3. Scenario Manager – Compare different sets of values in a model.

##### 1.1 Goal Seek

###### Definition:

Goal Seek is used when you know the desired result of a formula but need to find the correct input value.

**Syntax:**

- Navigate to Data → What-If Analysis → Goal Seek
- Set a Target Cell (Formula Cell)
- Enter the Desired Value
- Choose the Cell to Change

**Example:****Real-Life Scenario:**

Imagine you need a loan and want to determine what monthly payment will let you pay off the loan in 3 years instead of 5.

**Steps:**

1. Open Excel and enter the following data: Loan Amount: 10,000  
Interest Rate: 5%  
Term (Years): 5  
Monthly Payment: (Calculated using `=PMT(rate/12, term*12, -loan_amount)`)
2. Use Goal Seek to set Monthly Payment based on a 3-year loan term.
3. Excel will automatically adjust the payment amount to match the new term.

**1.2 Data Table****Definition:**

A Data Table is used to analyze multiple input values and see how they affect an output formula.

**Example:****Real-Life Scenario:**

You are an investor trying to see how different interest rates will affect future savings.

**Steps:**

1. Create a Principal Amount, Interest Rate, and Time Period table.
2. Use the FV function to calculate the future value.
3. Apply a Data Table to generate multiple results for different interest rates.

### **1.3 Scenario Manager**

**Definition:**

The Scenario Manager allows you to compare different sets of input values and analyze their impact on results.

**Example:**

**Real-Life Scenario:**

A business wants to compare Best Case, Worst Case, and Expected Revenue Scenarios.

**Steps:**

1. Create a table with variables like Sales, Expenses, and Profit.
2. Use Scenario Manager to create different scenarios with different values.
3. Excel will display all scenarios side by side for comparison.

## **2. Solver Add-in for Optimization Problems**

**What is Solver?**

Solver is an optimization tool that finds the best solution by adjusting multiple variables while meeting constraints (e.g., maximizing profit, minimizing cost).

**Example:**

**Real-Life Scenario:**

A factory produces two types of products using limited raw materials and labor hours. The goal is to maximize profit while staying within constraints.

**Steps:**

1. Enable Solver from File → Options → Add-ins → Manage Excel Add-ins
2. Define:
  - a. Objective: Maximize total profit
  - b. Decision Variables: Number of units of each product
  - c. Constraints: Limited raw materials and labor hours
3. Click Solve to find the optimal solution.

### **3. Data Analysis Toolpak**

**What is the Data Analysis Toolpak?**

The Data Analysis Toolpak contains statistical analysis tools like Regression, Descriptive Statistics, Histogram, and more.

#### **3.1 Regression Analysis**

**Definition:**

Regression determines relationships between independent and dependent variables.

**Example:**

**Real-Life Scenario:**

A sales manager wants to predict sales revenue based on advertising budget.

**Steps:**

1. Enable Data Analysis Toolpak from File → Options → Add-ins.
2. Select Regression and input Advertising Spend (X) and Sales Revenue (Y).
3. Excel will generate a regression model showing how advertising affects sales.

## **4. Trendlines and Forecasting**

### **What is a Trendline?**

A trendline shows patterns in data and helps predict future values.

### **Example:**

#### **Real-Life Scenario:**

An e-commerce store wants to predict next month's sales based on past trends.

**Steps:**

1. Create a chart with past sales data.
2. Click on the chart → Add Trendline.
3. Choose a trend type (Linear, Exponential, or Moving Average).
4. Extend the trendline to forecast future sales.

## **5. Correlation and Covariance**

### **5.1 Correlation**

#### **Definition:**

Correlation measures the strength of the relationship between two variables.

#### **Formula:**

=CORREL(array1, array2)

**Example:**

**Real-Life Scenario:**

A university wants to see if students' study hours correlate with exam scores.

**Steps:**

1. Input Study Hours in Column A and Exam Scores in Column B.
2. Use the formula: =CORREL(A2:A10, B2:B10)
3. A value close to 1 means strong correlation, close to 0 means no correlation.

## **5.2 Covariance**

**Definition:**

Covariance measures how two variables move together (positive or negative relationship).

**Formula:**

=COVARIANCE.P(array1, array2)

**Example:**

**Real-Life Scenario:**

A stock market analyst wants to analyze the relationship between two stock prices.

**Steps:**

1. Input Stock A prices in Column A and Stock B prices in Column B.
2. Use the formula: =COVARIANCE.P(A2:A10, B2:B10)
3. A positive value means they move together, a negative value means they move opposite.

## Conclusion

Advanced Data Analysis in Excel is a powerful tool for businesses, researchers, and analysts to make data-driven decisions. By mastering these techniques, you can optimize processes, forecast trends, and analyze relationships between key variables.

## Mini Project 1: Sales Forecast and Profit Optimization for an E-Commerce Business

### Project Requirement:

An e-commerce company wants to analyze its sales data to forecast revenue, optimize profits, and make data-driven decisions. The company will use What-If Analysis (Goal Seek, Data Tables, Scenario Manager), Solver for profit optimization, Regression for sales prediction, Trendlines for forecasting, and Correlation to analyze relationships between marketing spend and sales revenue.

### Step-by-Step Implementation

#### Step 1: Setting Up the Data

Create a table with the following data:

Month	Ad Spend (\$)	Units Sold	Price per Unit (\$)	Total Revenue (\$)	Profit (\$)
Jan	500	150	20	=C2*D2	=E2 - B2
Feb	700	180	22	=C3*D3	=E3 - B3
Mar	600	160	21	=C4*D4	=E4 - B4
...	...	...	...	...	...

## **Step 2: Using Goal Seek to Find the Required Ad Spend**

Problem Statement: The company wants to determine how much to spend on advertising in June to achieve a profit of \$5,000.

### **Steps:**

1. Go to Data → What-If Analysis → Goal Seek
2. Set Cell: Profit Cell for June
3. To Value: 5000
4. By Changing Cell: Ad Spend for June
5. Click OK – Excel will adjust Ad Spend to reach the profit goal.

## **Step 3: Using Data Table to Analyze Different Pricing Strategies**

Problem Statement: The company wants to analyze how different product prices affect total revenue.

### **Steps:**

1. Create a column with different Price per Unit values (e.g., \$18, \$20, \$22, etc.).
2. In the next column, reference the Total Revenue formula.
3. Select the Data Table range.
4. Go to Data → What-If Analysis → Data Table.
5. In the Column Input Cell, select the Price per Unit cell.
6. Click OK – Excel will generate revenue values for different pricing strategies.

## **Step 4: Using Scenario Manager for Best, Worst, and Expected Cases**

Problem Statement: The company wants to compare three scenarios: Best Case, Worst Case, and Expected Case.

**Steps:**

1. Go to Data → What-If Analysis → Scenario Manager
2. Click Add and enter the three scenarios:
  - a. Best Case: High Sales, Low Ad Spend
  - b. Worst Case: Low Sales, High Ad Spend
  - c. Expected Case: Moderate Sales and Ad Spend
3. Click Show to compare different scenarios.

**Step 5: Using Solver to Optimize Profit**

Problem Statement: The company wants to maximize profit by adjusting Ad Spend and Price per Unit while staying within a budget limit.

**Steps:**

1. Go to Data → Solver.
2. Set Objective: Maximize the Profit cell.
3. By Changing Variables: Ad Spend and Price per Unit.
4. Add Constraints:
  - a. Ad Spend  $\leq \$10,000$
  - b. Price per Unit between \$15 and \$30
5. Click Solve – Excel will suggest the best values to maximize profit.

**Step 6: Using Regression Analysis to Predict Sales**

Problem Statement: The company wants to predict future sales based on ad spend.

**Steps:**

1. Go to Data → Data Analysis → Regression.
2. Select Input Y Range (Units Sold).

3. Select Input X Range (Ad Spend).
4. Click OK – Excel will generate a regression model showing how ad spend affects sales.

### **Step 7: Using Trendlines for Forecasting**

Problem Statement: The company wants to predict next month's revenue based on past trends.

#### **Steps:**

1. Create a chart for revenue over time.
2. Click on the chart → Add Trendline.
3. Choose Linear or Exponential Trendline.
4. Extend the trendline to forecast future revenue.

### **Step 8: Using Correlation to Analyze the Impact of Marketing on Sales**

Problem Statement: The company wants to check if marketing spend is strongly correlated with sales growth.

#### **Steps:**

1. Use the CORREL function: =CORREL(B2:B10, C2:C10)
2. A value close to 1 means strong correlation, close to 0 means no correlation.

#### **Final Outcome:**

- Optimized Ad Spend for maximum profit
- Predicted Sales and Revenue
- Analyzed different pricing strategies
- Forecasted revenue trends
- Evaluated the impact of marketing on sales

## Mini Project 2: Inventory Optimization for a Retail Store

### Project Requirement:

A retail store wants to optimize inventory levels, forecast demand, and manage costs. It will use What-If Analysis, Solver, Regression, and Trendlines to analyze stock levels and sales trends.

### Step-by-Step Implementation

#### Step 1: Setting Up the Data

Create a table with the following data:

Product	Units in Stock	Demand Forecast	Reorder Point	Safety Stock	Total Cost
Item A	200	300	250	50	=D2*10
Item B	150	200	180	40	=D3*15
Item C	100	150	120	30	=D4*8

#### Step 2: Using Goal Seek to Find Reorder Levels

Problem Statement: Find the ideal reorder point to minimize shortages.

1. Use Goal Seek to adjust reorder points based on demand.
2. Set the Total Cost as the target cell.
3. Adjust Reorder Point to meet demand.

#### Step 3: Using Solver to Minimize Inventory Costs

Problem Statement: Optimize inventory costs while avoiding stockouts.

1. Set the Objective to minimize Total Cost.
2. Change Variables: Units in Stock and Safety Stock.

3. Add constraints:
  - a. Demand Forecast must be met.
  - b. Safety Stock must be  $\geq 10\%$  of demand.
4. Click **Solve** – Excel will suggest the best reorder levels.

#### **Step 4: Using Regression for Demand Forecasting**

Problem Statement: Predict future inventory demand based on past sales.

1. Use Regression Analysis in Data Analysis Toolpak.
2. Select past demand data as Y and time period as X.
3. Excel will generate a trendline equation to forecast future demand.

#### **Step 5: Using Trendlines for Future Inventory Planning**

Problem Statement: Visualize sales trends to plan inventory restocking.

1. Create a line chart for sales data.
2. Add a trendline to show demand trends.
3. Extend the trendline to forecast future sales.

#### **Final Outcome:**

- Optimized inventory costs
- Accurate demand forecasting
- Minimized stock shortages
- Improved inventory planning

## Conclusion

These projects demonstrate how Advanced Data Analysis in Excel can help businesses optimize profits, minimize costs, and forecast trends using real-life datasets.

## Real-Life Mini Project: Financial Planning and Investment Analysis

### Project Overview:

A financial analyst wants to help a client optimize their investment strategy by analyzing different scenarios for savings, investment returns, and retirement planning. The analyst will use What-If Analysis, Solver for portfolio optimization, Data Analysis Toolpak for statistical insights, and Trendlines for forecasting future financial growth.

## Day 8 Tasks

### 1. Set Up a Financial Dataset

- Create a dataset with income, expenses, savings, and investment details.
- Include monthly cash flow, interest rates, and risk factors.

### 2. Use Goal Seek to Determine Required Monthly Savings

- The client wants to save \$1,000,000 in 20 years.
- Use Goal Seek to calculate the required monthly savings at a 6% annual interest rate.

### 3. Create a Data Table for Different Interest Rates

- Show the effect of varying interest rates on final savings.

- Use a one-variable data table to analyze the impact of rates from 3% to 10%.

#### 4. Use Scenario Manager for Investment Strategies

- Compare Conservative, Moderate, and Aggressive investment approaches.
- Adjust risk levels and expected returns for each scenario.

#### 5. Optimize Portfolio Allocation Using Solver

- Allocate investments in stocks, bonds, and real estate to maximize returns while keeping risk below 10%.
- Use Solver to find the best allocation.

#### 6. Perform Regression Analysis to Predict Future Savings

- Use past income and savings data to forecast future savings trends.
- Apply Regression Analysis from the Data Analysis Toolpak.

#### 7. Analyze Monthly Expenses Using Descriptive Statistics

- Calculate mean, median, and standard deviation for different expense categories.
- Identify spending patterns using Data Analysis Toolpak.

#### 8. Add a Trendline for Investment Growth

- Create a line chart showing savings over time.
- Add a linear or exponential trendline to predict future savings.

#### 9. Forecast Retirement Savings Using Future Value Formula

- Calculate future savings based on current savings, monthly contributions, and expected return rates.

- Use the FV (Future Value) function in Excel.

#### 10. Compare Correlation Between Income and Savings

- Use the CORREL function to check if higher income leads to higher savings.
- Interpret the results to find financial habits.

#### 11. Apply Covariance Analysis on Investment Returns

- Compare historical stock and bond returns to determine diversification benefits.
- Use the COVARIANCE.P function in Excel.

#### 12. Visualize Savings & Investments with a Dashboard

- Create charts and graphs showing savings growth, investment allocation, and risk analysis.
- Use conditional formatting for insights.

#### 13. Provide a Final Financial Strategy Report

- Summarize findings, recommendations, and insights from the analysis.
- Suggest the best investment strategy for long-term growth.

### **Mini Project 1: Sales Forecasting & Pricing Strategy Optimization**

#### **Project Requirement:**

A retail company wants to forecast quarterly sales based on historical data and optimize its pricing strategy for maximum revenue. The project will involve:

- Using Goal Seek to determine the optimal product price for a target revenue.

- Creating a Data Table to analyze the impact of different discount rates on total sales.
- Using Scenario Manager to compare different sales growth scenarios.
- Applying Regression Analysis to predict future sales based on historical trends.
- Using Solver to optimize pricing and discount strategies while maximizing profits.
- Adding Trendlines to visualize expected sales growth over time.
- Using Correlation Analysis to understand the relationship between price changes and customer demand.

## **Mini Project 2: Budget Planning & Expense Optimization for a Business**

### **Project Requirement:**

A small business wants to analyze its monthly budget, optimize expenses, and forecast future financial health. The project will involve:

- Using What-If Analysis (Goal Seek) to determine required revenue for a profit target.
- Creating a Data Table to analyze cost variations with different marketing budgets.
- Using Scenario Manager to compare expense reduction strategies.
- Applying Solver to optimize cost allocation across departments.
- Performing Regression Analysis to predict future financial trends based on past spending.
- Adding Trendlines to visualize business growth trends over the next five years.
- Using Covariance Analysis to examine relationships between expenses (e.g., advertising vs. sales growth).

# Day 9

## Introduction to Macros in Excel

### What is a Macro?

A macro in Excel is a recorded sequence of actions that can be executed automatically to save time and effort. Instead of performing repetitive tasks manually, you can record and play back a macro to complete them instantly.

### Where Can You Find Macros in Excel?

- Open Excel → Go to the "Developer" tab.
- If the Developer tab is not visible:
  - Click File → Options → Customize Ribbon.
  - Check Developer → Click OK.
- Now, you can access Macros, VBA Editor, and other automation tools.

## Recording Macros

### Definition

A recorded macro captures mouse clicks, keystrokes, and formatting changes and allows you to replay them at any time.

### Step-by-Step Implementation

#### Example: Automating Cell Formatting using a Macro

Let's say you regularly format report headers (bold, red text, centered). Instead of doing this manually, we'll record a macro.

### Steps to Record a Macro

1. Open Excel and go to the Developer tab.
2. Click Record Macro.

3. In the pop-up window:
  - a. Macro name: FormatHeader
  - b. Shortcut key: (Optional, e.g., Ctrl + Shift + H)
  - c. Store macro in: This Workbook.
  - d. Click OK.
4. Perform the actions you want to record:
  - a. Select A1:D1 (header cells).
  - b. Click Bold (Ctrl + B).
  - c. Change text color to Red.
  - d. Center-align the text.
5. Stop recording:
  - a. Go to Developer tab → Click Stop Recording.

### **Running the Recorded Macro**

1. Go to Developer → Click Macros.
2. Select FormatHeader → Click Run.
3. The macro will apply the recorded formatting automatically.

### **Assigning Macros to Buttons**

You can create a button in Excel and assign a macro to it for quick execution.

#### **Steps to Assign a Macro to a Button**

1. Go to the Developer tab → Click Insert.
2. Under Form Controls, select Button.
3. Click anywhere on the sheet to place the button.
4. In the Assign Macro window, select FormatHeader → Click OK.
5. Change the button text to "Format Header".
6. Click the button, and it will run the macro instantly.

## Introduction to VBA (Visual Basic for Applications)

### What is VBA?

VBA (Visual Basic for Applications) is a programming language in Excel used for writing custom scripts to automate complex tasks that macros cannot record.

### Opening the VBA Editor

1. Go to Developer → Click Visual Basic.
2. The VBA Editor window will open.
3. Click Insert → Select Module to write VBA code.

## Writing Simple VBA Functions

### Example 1: Displaying a Message Box (Hello World)

#### Definition:

A simple VBA function that displays a pop-up message in Excel.

### Step-by-Step Implementation

1. Open the VBA Editor (Alt + F11).
2. Click Insert → Module.
3. Enter the following VBA code:

```
Sub HelloWorld()
    MsgBox "Hello, welcome to VBA in Excel!"
End Sub
```

4. Close the VBA Editor.
5. Go back to Excel → Press Alt + F8, select HelloWorld, and click Run.
6. A message box will pop up with the text "Hello, welcome to VBA in Excel!".

## Assigning and Running Simple VBA Scripts

### Example 2: Auto-Filling Data Using VBA

This VBA script will fill the first 10 rows of Column A with numbers 1 to 10.

### Step-by-Step Implementation

1. Open VBA Editor (Alt + F11).
2. Click Insert → Module.
3. Enter the following code:

```
Sub FillNumbers()
    Dim i As Integer
    For i = 1 To 10
        Cells(i, 1).Value = i
    Next i
End Sub
```

4. Close the VBA Editor.
5. Go back to Excel → Press Alt + F8, select FillNumbers, and click Run.
6. Column A will now be filled with numbers 1 to 10 automatically.

## Conclusion

Macros and VBA in Excel save time by automating repetitive tasks. You can record macros for simple automation or write VBA scripts for advanced customization. Learning these features helps in data entry, formatting, calculations, and complex workflows.

## Mini Project 1: Automated Report Formatting with Macros and VBA

### Project Requirement:

A company generates weekly sales reports, but the formatting process is repetitive. You need to create a **macro** that automates the following tasks:

1. Apply bold and red color to the headers.
2. Auto-adjust column widths.
3. Format sales figures as currency.
4. Assign this macro to a button for one-click formatting.

### Step-by-Step Implementation:

#### Step 1: Record a Macro for Formatting

1. Open Excel and navigate to the Developer tab.
2. Click Record Macro.
3. In the pop-up window:
  - a. Macro Name: FormatSalesReport
  - b. Shortcut Key (optional): Ctrl + Shift + F
  - c. Store macro in: This Workbook
  - d. Click OK.
4. Now, apply the formatting manually:
  - a. Select the header row (A1:E1) and make it bold and red.
  - b. Click Format Cells → Change number format to Currency for sales data.
  - c. Auto-adjust column width using Double-click on column separators.
5. Stop recording:
  - a. Go to Developer → Click Stop Recording.

#### Step 2: Assign the Macro to a Button

1. Go to Developer → Click Insert.

2. Under Form Controls, select Button.
3. Click anywhere on the sheet to place the button.
4. In the Assign Macro window, select FormatSalesReport → Click OK.
5. Rename the button to "Format Report".

### **Step 3: Running the Macro**

- Click the "Format Report" button.
- The macro automatically formats the report.

### **Step 4: VBA Code Explanation (Alternative to Recording a Macro)**

If you want to write the macro manually using VBA:

1. Open the VBA Editor (Alt + F11).
2. Click Insert → Select Module.
3. Enter the following code:

```
Sub FormatSalesReport()
    ' Apply bold and red color to headers
    Range("A1:E1").Font.Bold = True
    Range("A1:E1").Font.Color = RGB(255, 0, 0)

    ' Auto adjust column width
    Columns("A:E").AutoFit

    ' Format sales figures as currency
    Range("B2:E100").NumberFormat = "$#,##0.00"
End Sub
```

4. Close the VBA editor and run the macro using Alt + F8.

### **Project Outcome:**

- A formatted report in one click.
- Saves manual effort.
- Enhances data presentation.

## **Mini Project 2: Automated Attendance Tracker with Macros and VBA**

### **Project Requirement:**

A school needs an automated attendance tracker that:

1. Fills dates dynamically in Row 1.
2. Allows teachers to mark "P" (Present) or "A" (Absent).
3. Highlights absent students in red.
4. Uses a VBA script for auto-highlighting.

### **Step-by-Step Implementation:**

#### **Step 1: Set Up the Attendance Sheet**

1. Open **Excel** and create a table: A1: Student Name | B1: 01-Mar | C1: 02-Mar | D1: 03-Mar | ...  
A2: John | B2: P | C2: A | D2: P | ...  
A3: Sarah | B3: A | C3: P | D3: A | ...
2. Save the file as Macro-Enabled Workbook (.xlsm).

#### **Step 2: Record a Macro for Highlighting Absences**

1. Go to Developer → Click Record Macro.
2. Name it **HighlightAbsentees**.
3. Select the data range (e.g., B2:Z100).
4. Apply Conditional Formatting:

- a. Rule: If Cell Contains "A" → Fill Red.
5. Stop recording.

### **Step 3: Assign the Macro to a Button**

- Insert a button and assign the HighlightAbsentees macro.
- Clicking the button will highlight absent students.

### **Step 4: Writing the VBA Script for Automation**

1. Open VBA Editor (Alt + F11).
2. Click Insert → Module.
3. Enter this VBA code:

```
Sub HighlightAbsentees()
    Dim ws As Worksheet
    Set ws = ActiveSheet

    Dim cell As Range
    For Each cell In ws.Range("B2:Z100")
        If cell.Value = "A" Then
            cell.Interior.Color = RGB(255, 0, 0) ' Red color
        ElseIf cell.Value = "P" Then
            cell.Interior.Color = RGB(0, 255, 0) ' Green for Present
        End If
    Next cell
End Sub
```

4. Close the VBA Editor and run the script (Alt + F8).

### **Project Outcome:**

- A real-time attendance sheet.
- One-click highlighting of absences.
- Reduces manual work.

## **Mini Project: Automated Invoice Generator Using Macros and VBA**

### **Project Requirement:**

A business generates multiple invoices manually every day. You need to create an automated invoice generator using Excel Macros and VBA. The invoice should:

1. Auto-fill customer details based on selection.
2. Auto-calculate total cost with tax.
3. Generate a unique invoice number for each transaction.
4. Allow one-click printing of the invoice.

## **Day 9 Tasks**

### **Task 1: Setting Up the Invoice Template**

- Design an invoice format with fields like Invoice Number, Date, Customer Name, Product, Quantity, Price, Total, Tax, and Grand Total.

### **Task 2: Creating a Customer Drop-down List**

- Use Data Validation to create a drop-down list of customers in the invoice form.

### **Task 3: Using Macros to Auto-Fill Customer Details**

- When a customer is selected, their details (e.g., address, contact) should auto-fill from a database sheet.

#### Task 4: Auto-Populating Product Details

- When a product is selected, its price per unit should auto-fill in the invoice.

#### Task 5: Writing a VBA Script to Calculate Total Cost

- Create a VBA function that multiplies quantity by unit price and updates the total.

#### Task 6: Adding a Tax Calculation Function

- Implement a VBA function to apply tax (e.g., 10%) on the total amount.

#### Task 7: Generating a Unique Invoice Number

- Write a VBA script that automatically increments the invoice number for each new invoice.

#### Task 8: Assigning Macros to a Button ("Generate Invoice")

- Create a button that automatically fills invoice details when clicked.

#### Task 9: Creating a "Clear Form" Macro

- Write a macro that resets the invoice form for a new entry.

#### Task 10: Saving Invoices Automatically with a Macro

- Create a macro that saves each invoice as a separate Excel file or PDF.

#### Task 11: Assigning a Macro to Print the Invoice

- Create a Print Invoice button that runs a macro to print the invoice instantly.

### Task 12: Writing a Simple "Hello World" VBA Function

- Add a basic "Hello World" message box script to introduce VBA scripting.

### Task 13: Running the Macros and Testing the Functionality

- Test all macros and ensure the invoice generation is fully automated.

#### **Expected Outcome:**

- A fully automated invoice generator in Excel.
- One-click invoice creation, calculation, saving, and printing.
- Reduces manual errors and saves time.

## **Real-Life Mini Project 1: Automated Employee Attendance Tracker**

#### **Project Requirement:**

A company wants to automate employee attendance tracking using Excel Macros and VBA. The system should:

1. Record daily attendance with a button click.
2. Auto-fill dates and employee names from a database.
3. Generate monthly attendance reports for payroll processing.
4. Highlight absent employees automatically using conditional formatting.

## **Real-Life Mini Project 2: Automated Expense Management System**

#### **Project Requirement:**

A small business needs an automated expense management system to track and categorize expenses efficiently. The system should:

1. Allow users to enter expenses via a form.
2. Auto-categorize expenses (e.g., Rent, Utilities, Salaries).
3. Generate monthly expense reports using Macros.
4. Display total expenses per category using VBA calculations.

## Day 10

### Advanced Excel Features

#### 1. Using Power Query to Import and Transform Data

##### Definition:

Power Query is a powerful Excel tool used to import, clean, transform, and automate data processing from various sources such as databases, Excel files, and web pages.

##### Syntax:

Power Query does not require specific formulas like Excel functions. It uses a step-based interface. However, you can write M language for advanced transformations.

##### Real-Life Example: Importing and Cleaning Sales Data

##### Scenario:

A company receives sales data from different sources every month in an Excel file, but it contains unnecessary columns and incorrect formats. Power Query helps clean the data automatically.

## Step-by-Step Implementation

1. Open Power Query
  - a. Go to Data → Get Data → From File → From Workbook
  - b. Select the Excel file and Load it.
2. Remove Unnecessary Columns
  - a. Click on the column header → Press Delete.
3. Change Column Names and Data Types
  - a. Double-click on column names to rename them.
  - b. Change date columns to Date format.
4. Filter Data
  - a. Click the dropdown filter on any column to remove blank rows or incorrect data.
5. Load the Cleaned Data into Excel
  - a. Click Close & Load to send the cleaned data to a new sheet.

## 2. Introduction to Power Pivot

### Definition:

Power Pivot allows users to analyze large datasets efficiently by creating relationships between tables and performing calculations using DAX (Data Analysis Expressions).

### Real-Life Example: Creating a Sales Dashboard

#### Scenario:

A retail store wants to combine sales and customer data to analyze sales trends and customer behavior.

## Step-by-Step Implementation

1. Enable Power Pivot

- a. Go to File → Options → Add-ins → Select Power Pivot → Enable it.
2. Load Data into Power Pivot
  - a. Select your dataset → Click Power Pivot → Add to Data Model.
3. Create Relationships Between Tables
  - a. Click Manage Relationships → Connect Sales and Customer tables using Customer ID.
4. Create a DAX Measure for Total Sales

TotalSales = SUM(SalesTable[Revenue])

- a. This calculates total revenue from the SalesTable.
5. Insert a Pivot Table
  - a. Use the created data model to generate Pivot Tables and Charts.

### **3. Advanced Lookup: Using XLOOKUP and FILTER**

#### **Definition:**

- XLOOKUP: A modern alternative to VLOOKUP and HLOOKUP. It searches for a value in one column and returns a corresponding value from another column.
- FILTER: Returns an array that meets certain conditions.

#### **Syntax:**

- XLOOKUP: =XLOOKUP(lookup\_value, lookup\_array, return\_array, [if\_not\_found])
- FILTER: =FILTER(array, include, [if\_empty])

#### **Real-Life Example: Searching for Product Prices**

#### **Scenario:**

A store wants to quickly find the price of a product using XLOOKUP.

## Step-by-Step Implementation

### 1. Create a Product List Table

Product	Price
Laptop	50000
Mouse	1500
Keyboard	2000

### 2. Use XLOOKUP to Find Price

=XLOOKUP("Laptop", A2:A10, B2:B10, "Not Found")

a. This finds the price of "Laptop".

### 3. Use FILTER to Show Products Below ₹5000

=FILTER(A2:B10, B2:B10<5000, "No products found")

## 4. Working with Dynamic Arrays

### Definition:

Dynamic Arrays allow Excel to return multiple values in a single formula.

### Real-Life Example: Sorting a List of Employees by Salary

### Scenario:

A company wants to sort employees by salary dynamically.

## Step-by-Step Implementation

### 1. Create an Employee Salary Table

Employee	Salary
Alex	50000
Bob	40000
Charlie	60000

2. Use SORT to Sort Employees by Salary

=SORT(A2:B10, 2, -1)

a. This sorts the salary column in descending order.

3. Use UNIQUE to Get a List of Unique Job Titles

=UNIQUE(A2:A10)

a. This removes duplicate job titles.

## Conclusion

These advanced Excel features allow users to analyze large datasets, automate data cleaning, and improve decision-making. Power Query, Power Pivot, XLOOKUP, and Dynamic Arrays are essential for business intelligence and reporting.

## Mini Project 1: Sales Data Analysis and Reporting Using Advanced Excel Features

### Project Requirement

A retail company collects daily sales data from multiple stores. However, the data contains inconsistencies, missing values, and is spread across multiple files. The company needs a consolidated report with total sales, best-selling products, and a store-wise summary using Power Query, Power Pivot, XLOOKUP, and Dynamic Arrays.

## Step-by-Step Implementation

### Step 1: Import and Transform Data using Power Query

1. Load Multiple Sales Files into Power Query
  - a. Go to Data → Get Data → From File → From Folder.
  - b. Select the folder containing all the sales files.
  - c. Click Combine & Load to merge all files into one table.
2. Clean the Data in Power Query
  - a. Remove Empty Rows: Click on the column dropdown → Uncheck "Null" values.
  - b. Fix Column Names: Rename columns like Store ID, Product, Quantity, Revenue.
  - c. Change Data Types: Convert "Date" columns to Date format, "Revenue" to Currency.
3. Load the Cleaned Data into Excel
  - a. Click Close & Load → Load data into a new sheet.

### Step 2: Create Data Model in Power Pivot

1. Enable Power Pivot
  - a. Go to File → Options → Add-ins → Select Power Pivot and enable it.
2. Create a Data Model
  - a. Select the sales data → Click Power Pivot → Add to Data Model.
  - b. Import a separate Product Table (Product ID, Name, Category) into Power Pivot.
  - c. Establish a relationship between the Sales Table and Product Table using Product ID.
3. Create DAX Measures for Total Revenue and Best-Selling Product

TotalRevenue = SUM(Sales[Revenue])

BestSellingProduct = CALCULATE(MAX(Sales[Quantity]), ALLEXCEPT(Sales,

Sales[Product]))

- a. This calculates the total revenue and finds the most sold product.

### **Step 3: Perform Advanced Lookups using XLOOKUP**

1. Find the Revenue for a Specific Product

=XLOOKUP("Laptop", Sales[Product], Sales[Revenue], "Not Found")

- a. This retrieves the total revenue for Laptop.

2. Find Store-wise Revenue using XLOOKUP

=XLOOKUP("Store A", Sales[Store], Sales[Revenue], "Not Found")

### **Step 4: Use Dynamic Arrays for Advanced Reporting**

1. Get a Unique List of Stores

=UNIQUE(Sales[Store])

- a. This creates a dynamic list of store names.

2. Sort Products by Highest Revenue

=SORT(Sales[Product], Sales[Revenue], -1)

- a. This sorts products in descending order of revenue.

### **Final Outcome**

- A cleaned and consolidated dataset of sales.
- A Power Pivot model with relationships between sales and product data.
- A dashboard with XLOOKUP-based lookups for revenue analysis.
- A dynamic list of stores and top-selling products using Dynamic Arrays.

## Mini Project 2: Employee Performance Dashboard using Advanced Excel Features

### Project Requirement

A company tracks the performance of employees across multiple departments based on sales, customer feedback, and attendance. The HR team needs an interactive dashboard with Power Query for data cleaning, Power Pivot for relationships, and XLOOKUP for quick lookups.

### Step-by-Step Implementation

#### Step 1: Import and Clean Employee Data using Power Query

1. Load Employee Performance Data
  - a. Data Sources: Excel files with Employee ID, Name, Department, Sales, Feedback Score, Attendance.
  - b. Go to Data → Get Data → From File → From Workbook.
  - c. Select employee\_data.xlsx and load it into Power Query.
2. Clean the Data
  - a. Remove Duplicates: Select Employee ID → Remove Duplicates.
  - b. Filter Invalid Data: Remove records with negative sales or feedback below 0.
  - c. Rename Columns: Rename Emp\_ID → Employee ID, Dept → Department.
  - d. Load Data to Excel: Click Close & Load.

#### Step 2: Build a Data Model in Power Pivot

1. Create Power Pivot Relationships
  - a. Load two tables:
    - i. Employee Performance Data (Sales, Feedback, Attendance)
    - ii. Department Data (Department ID, Name, Manager)
  - b. Create a relationship using Department ID.

## 2. Create DAX Measures

- a. Total Sales by Employee TotalSales =  

$$\text{SUM}(\text{EmployeePerformance}[Sales])$$
  
- b. Average Feedback Score AvgFeedback =  

$$\text{AVERAGE}(\text{EmployeePerformance}[Feedback])$$
  
- c. Top Performer TopPerformer =  

$$\text{CALCULATE}(\text{MAX}(\text{EmployeePerformance}[Sales]), \text{ALLEXCEPT}(\text{EmployeePerformance}, \text{EmployeePerformance}[EmployeeID]))$$

## Step 3: Use XLOOKUP for Quick Analysis

### 1. Find the Sales of a Specific Employee

```
=XLOOKUP("John Doe", EmployeePerformance[Employee Name], EmployeePerformance[Sales], "Not Found")
```

### 2. Find Department Manager Based on Employee ID

```
=XLOOKUP(A2, EmployeePerformance[Employee ID], DepartmentData[Manager], "No Manager Found")
```

## Step 4: Use Dynamic Arrays for Reports

### 1. Get a List of Employees with High Sales

```
=FILTER(EmployeePerformance[Employee Name], EmployeePerformance[Sales]>50000, "No employees found")
```

- a. This filters employees who have more than ₹50,000 in sales.

## 2. Sort Employees by Attendance Score

```
=SORT(EmployeePerformance[Employee Name],  
EmployeePerformance[Attendance], -1)
```

- a. This sorts employees by attendance in descending order.

### Final Outcome

- A cleaned employee dataset with structured relationships in Power Pivot.
- A dashboard with performance metrics using DAX calculations.
- XLOOKUP-based searches to find employee sales and managers quickly.
- A dynamic ranking of employees using Dynamic Arrays.

### Conclusion

These two projects showcase real-world applications of Advanced Excel features, combining Power Query, Power Pivot, XLOOKUP, and Dynamic Arrays to create automated and insightful reports.

## Mini Project: Financial Data Analysis & Forecasting using Advanced Excel Features

### Project Goal:

A financial company needs to analyze historical revenue and expense data, build a financial forecast model, and generate insights using Power Query, Power Pivot, DAX, XLOOKUP, and Dynamic Arrays.

## Day 10 Tasks:

1. Import and Consolidate Data Using Power Query
  - a. Import multiple monthly financial reports from different files and merge them into a single dataset.
2. Data Cleaning and Transformation in Power Query
  - a. Remove duplicate entries, filter out null values, and standardize date formats.
3. Create a Financial Data Model in Power Pivot
  - a. Load cleaned data into Power Pivot and establish relationships between revenue, expense, and profit tables.
4. Use DAX to Calculate Key Financial Metrics
  - a. Create measures for Total Revenue, Total Expenses, Net Profit, and Profit Margin.
5. Generate Yearly and Quarterly Revenue Reports Using DAX
  - a. Use SUM, AVERAGE, and TIME Intelligence functions to analyze revenue trends over time.
6. Perform Advanced Lookups Using XLOOKUP
  - a. Fetch revenue details for a specific product or service using XLOOKUP.
7. Use FILTER to Extract High-Value Transactions
  - a. Identify and list transactions with revenue above a certain threshold using the FILTER function.
8. Create Dynamic Sales Forecasting Using Trend Analysis
  - a. Use FORECAST and TREND functions to project next quarter's revenue.
9. Build an Interactive Revenue Dashboard with Pivot Tables
  - a. Create dynamic pivot tables and charts to visualize revenue and expense patterns.
10. Use Dynamic Arrays to Automate Data Sorting

- Implement SORT and UNIQUE functions to list top-performing products dynamically.

#### 11. Generate a Dynamic Profitability Ranking Table

- Rank products/services based on profitability using RANK.EQ and LARGE functions.

#### 12. Implement Conditional Formatting for Financial Alerts

- Highlight profit margins below a specific percentage using conditional formatting.

#### 13. Create a Custom Financial Report Export Option

- Automate data extraction and report generation using Power Query for future use.

#### **Outcome:**

- A fully automated financial reporting system in Excel.
- Real-time data connections and insights using Power Pivot & DAX.
- Dynamic and interactive financial dashboards for business decision-making.

### **Mini Project 1: Employee Performance Dashboard using Advanced Excel Features**

#### **Project Requirement:**

A company wants to track employee performance metrics such as attendance, project completion, and customer feedback scores. The goal is to build an interactive dashboard using Power Query, Power Pivot, DAX calculations, XLOOKUP, and Dynamic Arrays to analyze and visualize employee performance data dynamically.

## Mini Project 2: E-Commerce Sales Analysis & Forecasting

### Project Requirement:

An e-commerce business needs to analyze historical sales data, track best-selling products, and forecast future sales. The project requires Power Query for data transformation, Power Pivot for data modeling, DAX for advanced calculations, and XLOOKUP for efficient product searching. Additionally, Dynamic Arrays will be used for sorting and filtering sales data dynamically.

## Day 11

### Excel Automation & Final Project: A Comprehensive Guide

#### 1. Automating Excel Tasks: Using Macros & VBA

##### Definition:

Excel Macros allow you to record and automate repetitive tasks, while VBA (Visual Basic for Applications) enables advanced automation using custom scripts.

##### Syntax (VBA Basics):

- Recording a Macro: No coding needed; Excel records actions.

- VBA Code Example: Sub HelloWorld()

```
    MsgBox "Hello, World!"
```

```
End Sub
```

This script displays a message box with "Hello, World!"

## Real-Life Example: Automating Monthly Report Formatting

### Step-by-Step Implementation:

1. Open Excel → Go to Developer Tab → Click Record Macro.
2. Perform Formatting Actions (e.g., bold headers, apply filters).
3. Stop Recording → Assign the Macro to a button for reuse.

## 2. Introduction to Power BI for Advanced Data Visualization

### Definition:

Power BI is a Microsoft tool that allows interactive data visualization and reporting.

### Steps to Use Power BI with Excel:

1. Export Data from Excel to Power BI:
  - a. Open Power BI → Click Get Data → Select Excel.
2. Create Visualizations:
  - a. Use Charts, Tables, and Maps to represent data.
3. Apply Filters & Slicers:
  - a. Enhance interactivity by adding slicers.

### Real-Life Example: Sales Dashboard

- Import Excel Sales Data → Create Sales by Region Chart → Add Filters for Date Ranges.

## 3. Collaboration Features: Sharing, Protecting Sheets, Track Changes

### Definition:

Excel allows multiple users to collaborate while protecting sensitive data.

### Key Features:

1. Sharing a Workbook:
  - a. File → Share → Add Emails.
2. Protecting a Sheet:

- a. Review Tab → Protect Sheet → Set a Password.
3. Tracking Changes:
  - a. Review Tab → Track Changes → Highlight modifications.

#### **Real-Life Example: Budget Approval Process**

- Finance team shares an Excel budget sheet while restricting editing to only authorized users.

### **4. Using Excel for Project Management (Gantt Charts, Timelines)**

#### **Definition:**

Excel helps in planning and tracking projects using tools like Gantt charts and timelines.

#### **Steps to Create a Simple Gantt Chart in Excel:**

1. List Project Tasks in Column A.
2. Enter Start & End Dates in Columns B & C.
3. Insert a Stacked Bar Chart:
  - a. Select Data → Insert Chart → Stacked Bar.
4. Format the Chart:
  - a. Hide the base bars, adjust colors, and add labels.

#### **Real-Life Example: Marketing Campaign Timeline**

- Create a Gantt chart to track social media and email marketing launch dates.

### **5. Final Project: Build an Interactive Dashboard**

#### **Definition:**

An interactive dashboard combines Pivot Tables, Charts, and Data Models to present data dynamically.

## **Step-by-Step Implementation:**

### **Step 1: Prepare Data**

- Collect sales data (Product, Region, Revenue, Date).
- Clean data using Power Query.

### **Step 2: Create Pivot Tables**

- Insert Pivot Table → Drag and drop fields.

### **Step 3: Add Pivot Charts**

- Convert Pivot Tables into Column, Pie, or Line Charts.

### **Step 4: Add Slicers & Filters**

- Insert Slicer to enable dynamic filtering.

### **Step 5: Design the Dashboard**

- Arrange elements, add titles, and format charts.

### **Real-Life Example: Company Sales Dashboard**

- Track sales by region, product, and time using interactive visualizations.

## **Conclusion**

By mastering Macros, Power BI, Collaboration, Project Management, and Dashboards, you can automate tasks, enhance teamwork, and visualize data effectively. These skills are crucial for business reporting, analysis, and project tracking.

## Mini Project 1: Automating Employee Attendance Report using Macros & VBA

### Project Requirement:

A company tracks employee attendance in an Excel sheet, but manually generating the monthly attendance summary is time-consuming. The goal is to automate this process using VBA macros, where:

1. The user inputs daily attendance (P for Present, A for Absent, L for Leave).
2. The VBA macro calculates attendance percentages per employee.
3. A button is provided to generate a summary report in a new sheet.

### Step-by-Step Implementation with Code Explanation

#### Step 1: Create the Attendance Data Sheet

- Open Excel and create a sheet named "Attendance" with the following format:

Employee ID	Name	01	02	...	30	Total Present	Attendance %
1001	John	P	A	...	P		
1002	Emma	P	P	...	A		

- Columns 01 to 30 represent days of the month with P (Present), A (Absent), L (Leave) entries.

#### Step 2: Open VBA Editor & Create a Macro

1. Press ALT + F11 to open the VBA Editor.
2. Click "Insert" → "Module" and paste the following VBA code:

```
Sub GenerateAttendanceSummary()
```

```
    Dim ws As Worksheet
```

```
    Dim lastRow As Integer
```

```
    Dim i As Integer
```

```

Dim totalDays As Integer
Dim presentCount As Integer

' Set the worksheet
Set ws = ThisWorkbook.Sheets("Attendance")

' Find the last row with data
lastRow = ws.Cells(Rows.Count, 1).End(xlUp).Row

' Define total working days
totalDays = 30 ' Adjust based on the month

' Loop through employees to calculate attendance
For i = 2 To lastRow
    presentCount = WorksheetFunction.CountIf(ws.Range(ws.Cells(i, 3),
ws.Cells(i, 2 + totalDays)), "P")
    ws.Cells(i, totalDays + 3).Value = presentCount
    ws.Cells(i, totalDays + 4).Value = Round((presentCount / totalDays) * 100, 2)
& "%"
Next i

MsgBox "Attendance summary updated!", vbInformation, "Success"
End Sub

```

### **Step 3: Assign Macro to a Button**

1. Go to "Developer" Tab → Insert a Button.
2. Right-click the Button → Assign Macro → Select "GenerateAttendanceSummary".
3. Click OK.

#### **Step 4: Run the Macro**

- Click the button, and it automatically calculates the total present days and attendance %.

#### **Expected Output:**

Employee ID	Name	01	02	...	30	Total Present	Attendance %
1001	John	P	A	...	P	26	86.67%
1002	Emma	P	P	...	A	23	76.67%

Automated attendance report without manual calculations!

### **Mini Project 2: Building an Interactive Sales Dashboard using Power BI & Excel**

#### **Project Requirement:**

A retail business wants to analyze monthly sales performance. The goal is to:

1. Import Excel sales data into Power BI.
2. Create an interactive dashboard with filters and charts.
3. Track product-wise and region-wise sales trends.

#### **Step-by-Step Implementation**

##### **Step 1: Prepare the Sales Data in Excel**

Create a sheet named "SalesData" with the following structure:

Date	Product	Category	Region	Units Sold	Revenue
01-Jan-24	Laptop	Gadgets	North	12	24000
03-Jan-24	Mouse	Gadgets	South	20	2000
05-Jan-24	Chair	Furniture	East	15	4500

Save this Excel file as SalesData.xlsx.

## Step 2: Import Data into Power BI

1. Open Power BI Desktop.
2. Click Home → Get Data → Excel and select SalesData.xlsx.
3. Click Load to import the dataset.

## Step 3: Create Interactive Visualizations

1. Create a Sales Trend Line Chart:
  - a. Click Insert → Line Chart.
  - b. Drag Date to X-axis and Revenue to Y-axis.
2. Create a Product Sales Bar Chart:
  - a. Click Insert → Bar Chart.
  - b. Drag Product to the X-axis and Units Sold to the Y-axis.
3. Add Region-wise Filters (Slicer):
  - a. Click Insert → Slicer and select Region.
  - b. Now, clicking a region will update all charts dynamically.

## Step 4: Format the Dashboard

- Add a title and background color.
- Adjust chart sizes & colors for readability.
- Add a Total Sales Card to show overall revenue.

## Step 5: Publish and Share

1. Click File → Publish → Power BI Service.
2. Share with team members via Power BI Online.

Now, users can filter and analyze sales data dynamically!

## Conclusion

- Project 1 (VBA Macros): Automated Employee Attendance Reports, reducing manual effort.
- Project 2 (Power BI): Created an interactive sales dashboard, improving decision-making.

## Mini Project: Automated Project Management Dashboard in Excel

### Project Objective:

Develop a fully automated Project Management Dashboard in Excel that helps track tasks, deadlines, team performance, and project progress using Macros, VBA, Pivot Tables, Charts, and Data Models.

## Day 11 Tasks

1. Set up a Project Tracking Table
  - a. Create a structured table with columns for Task Name, Assigned To, Start Date, End Date, Status, and Progress %.
2. Automate Task Status Updates with VBA
  - a. Use VBA macros to automatically update task statuses based on due dates (e.g., "On Track", "Delayed", "Completed").
3. Create a Gantt Chart for Project Timeline
  - a. Use Conditional Formatting to visualize project timelines dynamically.
4. Generate an Interactive Task Completion Report
  - a. Use Pivot Tables and Slicers to filter and display tasks by team members, departments, or priority.
5. Automate Team Performance Analysis
  - a. Use Power Query to transform and summarize task completion data by employee.

6. Protect Critical Sheets and Data
  - a. Implement Sheet Protection & Track Changes to prevent accidental data loss or unauthorized edits.
7. Create an Automated Task Reminder System
  - a. Use VBA Macros to trigger email notifications when deadlines are near.
8. Add a Power BI Dashboard Integration
  - a. Connect Excel data to Power BI and create visual reports for stakeholders.
9. Implement Dynamic XLOOKUP for Task Assignments
  - a. Use XLOOKUP and FILTER to dynamically fetch employee names based on task categories.
10. Build a Performance Scorecard Using Power Pivot
  - a. Use DAX formulas to calculate task efficiency scores per team member.
11. Automate Data Refreshing with VBA
  - a. Schedule automatic data updates for real-time project tracking.
12. Export Dashboard to PDF with a Macro
  - a. Create a VBA script to export reports and dashboard snapshots as PDF files.
13. Final Interactive Dashboard with Pivot Charts & Graphs
  - a. Build a Project Overview Dashboard that includes task status, resource utilization, and project health indicators.

## **Mini Project 1: Sales Performance Automation & Dashboard**

### **Project Requirement:**

Build an automated sales performance tracking system in Excel using Macros, VBA, Pivot Tables, and Power BI. The system should:

- Automatically import monthly sales data from multiple sources using Power Query.
- Use VBA macros to clean and standardize the data.
- Generate Pivot Charts and Interactive Dashboards for analyzing sales trends, revenue, and top-performing products.
- Implement dynamic filters using Slicers and Power Pivot.
- Integrate with Power BI to provide advanced visual analytics for stakeholders.
- Protect sensitive data with password protection and track changes.

## **Mini Project 2: Employee Attendance & Productivity Tracker**

### **Project Requirement:**

Develop a smart employee attendance and productivity monitoring system in Excel using Macros, Power Query, and VBA. The system should:

- Automate attendance data collection from multiple Excel sheets.
- Use VBA macros to generate automated reports on absenteeism and overtime.
- Apply Conditional Formatting and Pivot Tables to highlight productivity trends.
- Generate an interactive Gantt chart to track employee shifts and project assignments.
- Implement Dynamic Arrays, XLOOKUP, and FILTER functions for real-time data updates.
- Automate email alerts for late or absent employees using VBA.
- Export final reports as PDFs or Power BI Dashboards for management review.

# SQL

## Day 12

### Introduction to SQL and Database Concepts

#### What is a Database?

A database is an organized collection of data that can be easily accessed, managed, and updated. It is designed to store, retrieve, and manage large amounts of information. Databases are typically used for applications ranging from websites to enterprise software systems.

- **Example:** A library database that stores information about books, authors, and library members.

#### Types of Databases

There are two main types of databases:

1. **Relational Databases (SQL Databases):**
  - a. These databases store data in tables (rows and columns) that are related to one another. The most common relational database management system (RDBMS) is **MySQL**.
  - b. Examples: MySQL, PostgreSQL, Oracle, Microsoft SQL Server.
2. **Non-relational Databases (NoSQL Databases):**
  - a. These databases store data in formats other than tables, such as key-value pairs, documents, or graphs. NoSQL databases are often used for large-scale applications or applications that require flexible schema design.
  - b. Examples: MongoDB, Redis, Cassandra, Firebase.

## Overview of SQL and its Role in Querying Databases

**SQL (Structured Query Language)** is a standard programming language used to manage and manipulate relational databases. SQL allows users to create, update, delete, and query data in relational databases. It acts as a bridge between the user and the database, providing commands for interacting with the data stored in relational tables.

## SQL Basics

### Understanding Relational Database Tables

A relational database table is a collection of data organized into rows (also called records) and columns (also called fields). Each column contains data of a specific type, and each row represents a single record in the database.

#### Example:

BookID	Title	Author	Year
1	To Kill a Mockingbird	Harper Lee	1960
2	1984	George Orwell	1949

### Basic SQL Syntax and Structure

SQL commands have a specific syntax, and they generally consist of the following structure:

- Command (e.g., SELECT)
- Target (e.g., table name or columns)
- Condition (e.g., WHERE clause)

### Key SQL Commands

1. **SELECT**: Retrieves data from one or more tables.

- a. Syntax: SELECT column1, column2, ... FROM table\_name;
  - b. Example: SELECT Title, Author FROM Books;
  - c. This retrieves the Title and Author columns from the Books table.
2. **FROM:** Specifies the table from which data should be retrieved.
- a. Already included in the SELECT statement.
3. **WHERE:** Filters the results based on a condition.
- a. Syntax: SELECT column1, column2 FROM table\_name WHERE condition;
  - b. Example: SELECT \* FROM Books WHERE Author = 'George Orwell';
  - c. This retrieves all columns (\*) from the Books table where the Author is 'George Orwell'.
4. **ORDER BY:** Sorts the result set.
- a. Syntax: SELECT column1, column2 FROM table\_name ORDER BY column1 [ASC|DESC];
  - b. Example: SELECT Title, Year FROM Books ORDER BY Year DESC;
  - c. This retrieves the Title and Year from the Books table, sorted by Year in descending order.
5. **LIMIT:** Restricts the number of rows returned.
- a. Syntax: SELECT column1, column2 FROM table\_name LIMIT number;
  - b. Example: SELECT \* FROM Books LIMIT 5;
  - c. This retrieves the first 5 rows from the Books table.

## Setting Up a Database

### Introduction to Database Management Systems (DBMS)

A DBMS (Database Management System) is software used to create, manage, and interact with databases. It provides an interface for users to define, query, and update the database.

Common DBMS software includes:

- MySQL: Popular open-source RDBMS.
- PostgreSQL: Advanced open-source RDBMS.
- SQLite: Lightweight, file-based RDBMS used for small applications.

Installing and Setting Up MySQL, PostgreSQL, or SQLite

Here are the steps for installing **MySQL**:

**1. Install MySQL:**

- a. On Windows: Use the MySQL installer from the [official website](#).
- b. On macOS: Use Homebrew: brew install mysql
- c. On Linux: Use the package manager (e.g., apt for Ubuntu): sudo apt-get install mysql-server

**2. Start MySQL Server:**

- a. On Windows and macOS, MySQL typically starts automatically after installation.
- b. On Linux: sudo systemctl start mysql

**3. Access MySQL:**

- a. Open a terminal or command prompt and type: mysql -u root -p
- b. Enter your password to access the MySQL shell.

Creating a Database and Tables

**1. Create a Database:**

- a. **Syntax:** CREATE DATABASE database\_name;
- b. **Example:** CREATE DATABASE Library;

## 2. Create a Table:

a. **Syntax:** CREATE TABLE table\_name (  
    column1 datatype,  
    column2 datatype,  
    ...  
);

b. **Example:** CREATE TABLE Books (  
    BookID INT PRIMARY KEY,  
    Title VARCHAR(255),  
    Author VARCHAR(255),  
    Year INT  
);

## Inserting Data into Tables (INSERT INTO)

### 1. Inserting Single Record:

a. **Syntax:** INSERT INTO table\_name (column1, column2, ...)  
    VALUES (value1, value2, ...);  
b. **Example:** INSERT INTO Books (BookID, Title, Author, Year)  
    VALUES (1, 'To Kill a Mockingbird', 'Harper Lee', 1960);

### 2. Inserting Multiple Records:

a. **Syntax:** INSERT INTO table\_name (column1, column2, ...)  
    VALUES (value1, value2, ...),  
              (value1, value2, ...),  
              ...;  
b. **Example:** INSERT INTO Books (BookID, Title, Author, Year)  
    VALUES (2, '1984', 'George Orwell', 1949),  
              (3, 'Moby Dick', 'Herman Melville', 1851);

## Mini Project 1: Library Management System (Relational Database)

### Overview:

This mini-project simulates a Library Management System using SQL, where users can manage books, authors, and members. It focuses on relational databases, where data is organized into different tables and connected using relationships.

### Steps:

#### 1. Database Setup:

- Install and set up **MySQL** or **SQLite**.
- Create a database called Library.

#### 2. Create Tables:

- Books**: Stores information about books in the library.
- Authors**: Stores information about authors.
- Members**: Stores information about library members.
- Loans**: Stores the loan history (which member borrowed which book and when).

### Example SQL:

```
CREATE DATABASE Library;
```

```
USE Library;
```

```
CREATE TABLE Authors (
    AuthorID INT PRIMARY KEY AUTO_INCREMENT,
    Name VARCHAR(255),
    BirthYear INT
);
```

```
CREATE TABLE Books (
    BookID INT PRIMARY KEY AUTO_INCREMENT,
    Title VARCHAR(255),
    AuthorID INT,
    Year INT,
    Genre VARCHAR(100),
    FOREIGN KEY (AuthorID) REFERENCES Authors(AuthorID)
);
```

```
CREATE TABLE Members (
    MemberID INT PRIMARY KEY AUTO_INCREMENT,
    Name VARCHAR(255),
    Email VARCHAR(255)
);
```

```
CREATE TABLE Loans (
    LoanID INT PRIMARY KEY AUTO_INCREMENT,
    BookID INT,
    MemberID INT,
    LoanDate DATE,
    ReturnDate DATE,
    FOREIGN KEY (BookID) REFERENCES Books(BookID),
    FOREIGN KEY (MemberID) REFERENCES Members(MemberID)
);
```

3. **Insert Data:** Add data to the Books, Authors, Members, and Loans tables.

```
INSERT INTO Authors (Name, BirthYear) VALUES ('Harper Lee', 1926);
INSERT INTO Books (Title, AuthorID, Year, Genre) VALUES ('To Kill a Mockingbird',
1, 1960, 'Fiction');
INSERT INTO Members (Name, Email) VALUES ('John Doe',
'john.doe@email.com');
INSERT INTO Loans (BookID, MemberID, LoanDate, ReturnDate) VALUES (1, 1,
'2025-03-01', '2025-03-15');
```

#### 4. SQL Queries:

- a. Retrieve all books by a specific author: `SELECT Title FROM Books WHERE AuthorID = 1;`
  
- b. Get all books borrowed by a specific member:  
`SELECT Books.Title,  
Loans.LoanDate, Loans.ReturnDate  
FROM Loans  
JOIN Books ON Loans.BookID = Books.BookID  
WHERE Loans.MemberID = 1;`

5. **Goal:** This project will help you understand relational databases, foreign keys, joins, and how to interact with different tables using SQL queries.

## Mini Project 2: Employee Management System (Relational Database)

### Overview:

This mini-project simulates an Employee Management System where you can track employee details, departments, and salaries. It focuses on creating a database, inserting data, and querying data using SQL.

Steps:

**1. Database Setup:**

- a. Install **MySQL** or **SQLite** and set up a new database called EmployeeDB.

**2. Create Tables:**

- a. **Employees:** Stores employee information like ID, name, salary, and department.
- b. **Departments:** Stores department details like department ID and name.
- c. **Salaries:** Stores salary history for employees.

**Example SQL:**

```
CREATE DATABASE EmployeeDB;
```

```
USE EmployeeDB;
```

```
CREATE TABLE Departments (
    DeptID INT PRIMARY KEY AUTO_INCREMENT,
    DeptName VARCHAR(255)
);
```

```
CREATE TABLE Employees (
    EmpID INT PRIMARY KEY AUTO_INCREMENT,
    Name VARCHAR(255),
    DeptID INT,
    HireDate DATE,
    FOREIGN KEY (DeptID) REFERENCES Departments(DeptID)
);
```

```

CREATE TABLE Salaries (
    SalaryID INT PRIMARY KEY AUTO_INCREMENT,
    EmpID INT,
    SalaryAmount DECIMAL(10, 2),
    EffectiveDate DATE,
    FOREIGN KEY (EmpID) REFERENCES Employees(EmpID)
);

```

**3. Insert Data:** Insert sample data for departments, employees, and salaries.

```

INSERT INTO Departments (DeptName) VALUES ('HR'), ('IT'), ('Sales');
INSERT INTO Employees (Name, DeptID, HireDate) VALUES ('Alice Smith', 1, '2020-01-15');
INSERT INTO Salaries (EmpID, SalaryAmount, EffectiveDate) VALUES (1, 55000.00, '2025-03-01');

```

**4. SQL Queries:**

- Get all employees in the IT department: `SELECT Name FROM Employees WHERE DeptID = (SELECT DeptID FROM Departments WHERE DeptName = 'IT');`
- Retrieve the current salary of a specific employee: `SELECT Employees.Name, Salaries.SalaryAmount FROM Salaries JOIN Employees ON Salaries.EmpID = Employees.EmpID WHERE Employees.Name = 'Alice Smith' ORDER BY Salaries.EffectiveDate DESC LIMIT 1;`

5. **Goal:** This project will help you practice using SQL commands such as SELECT, JOIN, WHERE, LIMIT, and ORDER BY, as well as creating relationships between tables using foreign keys.

## Day 12 Tasks :

### 1. Understanding What a Database Is

- Define what a database is in simple terms. Describe how a database is used to store, manage, and retrieve data.

### 2. Types of Databases (Relational vs. Non-relational)

- Write a comparison between relational databases and non-relational databases. Include examples of each type.

### 3. Overview of SQL and Its Role

- Explain the role of SQL in querying and manipulating databases. Discuss its major components like DDL (Data Definition Language) and DML (Data Manipulation Language).

### 4. Understanding Relational Database Tables

- Describe what relational database tables are and how they store data in rows and columns. Explain the concept of primary keys and foreign keys.

### 5. Basic SQL Syntax and Structure

- Write a short explanation of SQL syntax, including common components like keywords, operators, and clauses.

## 6. The SELECT Command

- Write a SQL query using the SELECT command to retrieve specific columns (e.g., name, position) from a table (e.g., employees table).

## 7. The FROM Clause

- Write a SQL query that uses the FROM clause to select data from a specified table.

## 8. The WHERE Clause

- Write a SQL query to retrieve data from a table based on a specific condition (e.g., retrieving all employees with a salary above a certain value).

## 9. The ORDER BY Clause

- Write a SQL query that sorts data by a column (e.g., salary) in ascending or descending order.

## 10. The LIMIT Clause

- Write a SQL query that retrieves a limited number of rows from a table, such as retrieving only the top 5 highest-paid employees.

## 11. Installing and Setting Up a Database Management System (DBMS)

- Install MySQL, PostgreSQL, or SQLite on your machine. Follow the installation steps and ensure the DBMS is running properly.

## 12. Creating a Database and Tables

- Create a new database and define tables within the database for a chosen system, such as a **Library Management System** or a **Customer Relationship Management (CRM)** system.

## 13. Inserting Data into Tables (INSERT INTO)

- Insert sample data into a table that you created in the previous task. Make sure to add multiple rows of data with different values.

## Mini Project 1: Online Store Product Catalog (Relational Database)

- **Objective:** Build a simple **Online Store Product Catalog** using a relational database to manage product details, categories, and stock availability.
- **Tasks:**
  - **Set up the Database:** Install MySQL or PostgreSQL and create a new database named OnlineStore.
  - **Create Tables:**
    - Create a products table with columns such as product\_id, name, category\_id, price, stock\_quantity.
    - Create a categories table with columns such as category\_id, category\_name.
  - **Insert Data:** Insert sample data into the products and categories tables (e.g., products like laptops, phones, and accessories, and corresponding categories).
  - **Query Data:**
    - Write a SQL query to list all products along with their category name.
    - Write a query to find all products under a specific category (e.g., Electronics).

- Write a query to find products that are out of stock (stock\_quantity = 0).
- Use SELECT, FROM, WHERE, and JOIN commands to retrieve relevant data.
- **Sort and Filter Data:**
  - Sort products by price in descending order.
  - Filter products by category and price range (e.g., products under \$500).
- **Limit Results:** Use the LIMIT clause to show only the top 10 most expensive products.

This updated mini-project focuses on an **Online Store Product Catalog**, where SQL commands are used to manage products, filter by category, check stock availability, and sort product data by price.

## Mini Project 2: Customer Relationship Management (CRM) System

- **Objective:** Create a **CRM System** to manage customer details and their orders.
- **Tasks:**
  - **Set up the Database:** Install SQLite or PostgreSQL and create a database named CRMSystem.
  - **Create Tables:** Define tables for customers (e.g., customer\_id, name, email, phone), orders (e.g., order\_id, customer\_id, order\_date), and order items (e.g., product\_name, quantity, price).
  - **Insert Data:** Insert sample customer and order data into the CRM tables.
  - **Query Data:**
    - Write a query to list all customers who have placed an order in the last month.

- Write a query to find all orders for a specific customer.
- Write a query to find the total amount spent by each customer by joining orders and order items tables.
- **Sort and Filter Data:** Implement sorting (e.g., by order date) and filtering (e.g., orders above a specific total price) using ORDER BY and WHERE.
- **Limit Results:** Use LIMIT to show only the top 3 customers with the highest spending.

## Day 13

### Data Retrieval and Filtering in SQL

#### 1. SELECT Statements

- **Definition:** The SELECT statement is used to retrieve data from a database. You can retrieve data from one or more tables in a database, and you can also specify which columns you want to retrieve.

#### Retrieving Data from a Single Table Using SELECT

- **Syntax:** `SELECT * FROM table_name;`
  - The \* retrieves all columns from the specified table.
  - Example: `SELECT * FROM employees;`  
This will retrieve all data from the employees table.

## Selecting Specific Columns

- **Syntax:** SELECT column1, column2 FROM table\_name;
  - This retrieves only the specified columns.
  - Example: SELECT name, salary FROM employees;  
This will retrieve the name and salary columns from the employees table.

## Using DISTINCT to Eliminate Duplicate Records

- **Syntax:** SELECT DISTINCT column1 FROM table\_name;
  - The DISTINCT keyword is used to return only unique (non-duplicate) values from the specified column.
  - Example: SELECT DISTINCT department FROM employees;  
This will retrieve a list of unique departments from the employees table.

## 2. Filtering Data

- **Definition:** Filtering in SQL allows you to retrieve specific records based on certain conditions. This is done using the WHERE clause with various operators.

## Filtering Records Using the WHERE Clause

- **Syntax:** SELECT column1, column2 FROM table\_name WHERE condition;
  - Example: SELECT name, salary FROM employees WHERE department = 'Sales';  
This will retrieve all employees who work in the "Sales" department.

## Comparison Operators

- **Syntax:** SELECT column1 FROM table\_name WHERE column1 operator value;
- Common Comparison Operators:
  - = (Equal to)
  - != (Not equal to)
  - < (Less than)
  - > (Greater than)
  - <= (Less than or equal to)
  - >= (Greater than or equal to)
- Example: SELECT name FROM employees WHERE salary > 50000;  
This retrieves all employees with a salary greater than 50,000.

## Logical Operators

- **Definition:** Logical operators are used to combine multiple conditions in a WHERE clause.
  - **AND:** Both conditions must be true.
  - **OR:** Either of the conditions can be true.
  - **NOT:** Reverses the condition (filters out the specified condition).
- **Syntax:**

SELECT column1 FROM table\_name WHERE condition1 AND/OR condition2;

- Example: SELECT name FROM employees WHERE salary > 50000 AND department = 'Sales';  
This will retrieve all employees in the "Sales" department with a salary greater than 50,000.

## Pattern Matching with LIKE (Wildcards % and \_)

- **Definition:** The LIKE operator is used to search for a specified pattern in a column. Wildcards are used to represent one or more characters.
  - %: Represents zero or more characters.
  - \_: Represents exactly one character.
- **Syntax:**

`SELECT column1 FROM table_name WHERE column1 LIKE pattern;`

- Example:

`SELECT name FROM employees WHERE name LIKE 'J%';`

This will retrieve all employees whose names start with "J".

`SELECT name FROM employees WHERE name LIKE 'J_n';`

This will retrieve all employees whose names are three characters long and start with "J" and end with "n" (e.g., "Jon").

## Using BETWEEN and IN for Range and List Filtering

- **BETWEEN:** The BETWEEN operator is used to filter records within a specific range (inclusive).
  - Syntax: `SELECT column1 FROM table_name WHERE column1 BETWEEN value1 AND value2;`
  - Example: `SELECT name, salary FROM employees WHERE salary BETWEEN 40000 AND 60000;`  
This retrieves employees with a salary between 40,000 and 60,000.

- **IN:** The IN operator is used to filter records that match any value from a list of specified values.
  - Syntax: `SELECT column1 FROM table_name WHERE column1 IN (value1, value2, ...);`
  - Example: `SELECT name FROM employees WHERE department IN ('Sales', 'Marketing');`  
This retrieves all employees who work in the "Sales" or "Marketing" departments.

### **NULL Handling (IS NULL, IS NOT NULL)**

- **Definition:** NULL represents an unknown or missing value. Use IS NULL or IS NOT NULL to check for NULL values.
- **Syntax:** `SELECT column1 FROM table_name WHERE column1 IS NULL;`
  - Example: `SELECT name FROM employees WHERE department IS NULL;`  
This retrieves all employees who do not belong to any department (NULL value).

### 3. Sorting Data

- **Definition:** Sorting allows you to arrange the result set in a specific order based on one or more columns.

### **Sorting Results Using ORDER BY**

- **Syntax:** `SELECT column1, column2 FROM table_name ORDER BY column1 [ASC|DESC];`
  - **ASC:** Sorts in ascending order (default).

- **DESC:** Sorts in descending order.
- Example: `SELECT name, salary FROM employees ORDER BY salary DESC;`  
This retrieves all employees and sorts them by salary in descending order (highest salary first).

## Sorting by Multiple Columns

- **Syntax:** `SELECT column1, column2 FROM table_name ORDER BY column1 [ASC|DESC], column2 [ASC|DESC];`
  - Example: `SELECT name, department, salary FROM employees ORDER BY department ASC, salary DESC;`  
This retrieves all employees, sorts them by department in ascending order, and within each department, sorts by salary in descending order.

## Summary:

- **Data Retrieval:** Use `SELECT` to fetch data from tables. Use `DISTINCT` to remove duplicates.
- **Filtering Data:** Use the `WHERE` clause with comparison operators (e.g., `=`, `>`, `<=`) and logical operators (e.g., `AND`, `OR`, `NOT`) to filter the data.
- **Pattern Matching:** Use `LIKE` with wildcards (`%`, `_`) for pattern matching.
- **Range and List Filtering:** Use `BETWEEN` for ranges and `IN` for multiple values.
- **NULL Handling:** Use `IS NULL` or `IS NOT NULL` to check for missing values.
- **Sorting Data:** Use `ORDER BY` to sort results and `ASC` or `DESC` for ascending or descending order. You can also sort by multiple columns.

## Mini Project 1: Employee Database – Retrieving and Filtering Employee Data

### Project Overview:

You will create a database for an employee management system where you can retrieve and filter employee data. You will perform the following tasks:

1. Retrieving employee names, departments, and salaries.
2. Filtering employees by salary range, department, and name.
3. Sorting employee data by salary in descending order and department in ascending order.

### Step-by-Step Solution:

#### **Step 1: Create the Employees Table**

```
CREATE DATABASE EmployeeDB;  
USE EmployeeDB;
```

```
CREATE TABLE employees (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    name VARCHAR(50),  
    department VARCHAR(50),  
    salary DECIMAL(10, 2)  
);
```

## **Step 2: Insert Sample Employee Data**

```
INSERT INTO employees (name, department, salary) VALUES  
('John Doe', 'Sales', 55000),  
('Jane Smith', 'Marketing', 60000),  
('Michael Brown', 'Sales', 45000),  
('Emily Johnson', 'HR', 52000),  
('David Lee', 'Marketing', 70000),  
('Sophia Davis', 'Sales', 48000);
```

## **Step 3: Retrieve All Employees**

To retrieve all employee data from the table:

```
SELECT * FROM employees;
```

## **Step 4: Select Specific Columns**

To select only the name and salary of all employees:

```
SELECT name, salary FROM employees;
```

## **Step 5: Use DISTINCT to Remove Duplicates**

To get a unique list of departments:

```
SELECT DISTINCT department FROM employees;
```

## **Step 6: Filtering Employees by Salary**

To retrieve employees whose salary is greater than 50,000:

```
SELECT name, salary FROM employees WHERE salary > 50000;
```

### **Step 7: Filtering Employees by Department**

To retrieve all employees working in the 'Sales' department:

```
SELECT name FROM employees WHERE department = 'Sales';
```

### **Step 8: Using Comparison Operators (<=)**

To retrieve employees with a salary less than or equal to 50,000:

```
SELECT name, salary FROM employees WHERE salary <= 50000;
```

### **Step 9: Filtering Employees Using Logical Operators (AND, OR)**

To retrieve employees working in 'Sales' and earning more than 50,000:

```
SELECT name FROM employees WHERE department = 'Sales' AND salary > 50000;
```

To retrieve employees working in either 'Sales' or 'Marketing':

```
SELECT name FROM employees WHERE department = 'Sales' OR department = 'Marketing';
```

### **Step 10: Pattern Matching with LIKE**

To find employees whose name starts with 'J':

```
SELECT name FROM employees WHERE name LIKE 'J%';
```

To find employees whose name contains 'an':

```
SELECT name FROM employees WHERE name LIKE '%an%';
```

### **Step 11: Using BETWEEN for Salary Range**

To find employees whose salary is between 50,000 and 70,000:

```
SELECT name, salary FROM employees WHERE salary BETWEEN 50000 AND  
70000;
```

### **Step 12: Filtering NULL Values**

Assume that the department column has some NULL values. To retrieve employees with no department assigned:

```
SELECT name FROM employees WHERE department IS NULL;
```

### **Step 13: Sorting the Results**

To sort employees by salary in descending order:

```
SELECT name, salary FROM employees ORDER BY salary DESC;
```

To sort employees by department in ascending order and salary in descending order:

```
SELECT name, department, salary FROM employees ORDER BY department ASC,  
salary DESC;
```

## Mini Project 2: Product Database – Retrieving and Filtering Product Data

Project Overview:

You will create a database for a product catalog. This project will help you practice retrieving and filtering product data.

1. Retrieve product names, categories, and prices.
2. Filter products by category, price range, and availability.
3. Sort products by price.

Step-by-Step Solution:

### Step 1: Create the Products Table

```
CREATE DATABASE ProductCatalog;  
USE ProductCatalog;
```

```
CREATE TABLE products (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    name VARCHAR(100),  
    category VARCHAR(50),  
    price DECIMAL(10, 2),  
    availability BOOLEAN  
);
```

### Step 2: Insert Sample Product Data

```
INSERT INTO products (name, category, price, availability) VALUES  
('Laptop', 'Electronics', 899.99, TRUE),  
('Wireless Mouse', 'Electronics', 29.99, TRUE),
```

```
('Office Chair', 'Furniture', 149.99, FALSE),  
('Smartphone', 'Electronics', 499.99, TRUE),  
('Desk Lamp', 'Furniture', 39.99, TRUE),  
('Gaming Keyboard', 'Electronics', 79.99, TRUE);
```

### **Step 3: Retrieve All Products**

To retrieve all product data from the table:

```
SELECT * FROM products;
```

### **Step 4: Select Specific Columns**

To retrieve only the name and price of all products:

```
SELECT name, price FROM products;
```

### **Step 5: Use DISTINCT to Remove Duplicates**

To get a unique list of product categories:

```
SELECT DISTINCT category FROM products;
```

### **Step 6: Filtering Products by Price**

To retrieve products that are priced less than 100:

```
SELECT name, price FROM products WHERE price < 100;
```

### **Step 7: Filtering Products by Category**

To retrieve all products in the 'Electronics' category:

```
SELECT name FROM products WHERE category = 'Electronics';
```

### **Step 8: Using Comparison Operators (>=)**

To retrieve products priced at 100 or more:

```
SELECT name, price FROM products WHERE price >= 100;
```

### **Step 9: Filtering Products Using Logical Operators (AND, OR)**

To retrieve products in the 'Furniture' category that are available:

```
SELECT name FROM products WHERE category = 'Furniture' AND availability = TRUE;
```

To retrieve products that are either available or in the 'Electronics' category:

```
SELECT name FROM products WHERE availability = TRUE OR category = 'Electronics';
```

### **Step 10: Pattern Matching with LIKE**

To find products whose names contain the word 'Smart':

```
SELECT name FROM products WHERE name LIKE '%Smart%';
```

### **Step 11: Using BETWEEN for Price Range**

To find products priced between 30 and 100:

```
SELECT name, price FROM products WHERE price BETWEEN 30 AND 100;
```

## Step 12: Filtering NULL Values

Assume that some products have missing availability status (NULL). To retrieve products where availability is NULL:

```
SELECT name FROM products WHERE availability IS NULL;
```

## Step 13: Sorting the Results

To sort products by price in ascending order:

```
SELECT name, price FROM products ORDER BY price ASC;
```

To sort products by category in ascending order and price in descending order:

```
SELECT name, category, price FROM products ORDER BY category ASC, price DESC;
```

## Summary of Tasks:

1. Create a database and tables.
2. Insert sample data into the tables.
3. Retrieve data from the table using SELECT.
4. Filter data using the WHERE clause and various operators (comparison, logical).
5. Handle NULL values using IS NULL and IS NOT NULL.
6. Use DISTINCT to remove duplicate values.
7. Filter data using pattern matching (LIKE).
8. Sort data using ORDER BY and different sorting orders.

These projects help you understand the basic SQL operations for retrieving and filtering data, which are crucial for building any database-driven application.

## Dataset: Employee Table

<b>id</b>	<b>name</b>	<b>department</b>	<b>salary</b>	<b>joining_date</b>	<b>email</b>	<b>manager_id</b>
1	John Doe	Sales	55000	2021-03-01	<a href="mailto:john.doe@email.com">john.doe@email.com</a>	NULL
2	Jane Smith	Marketing	60000	2020-01-15	<a href="mailto:jane.smith@email.com">jane.smith@email.com</a>	1
3	Michael Brown	Sales	45000	2021-07-25	<a href="mailto:michael.brown@email.com">michael.brown@email.com</a>	1
4	Emily Johnson	HR	52000	2019-10-10	<a href="mailto:emily.johnson@email.com">emily.johnson@email.com</a>	NULL
5	David Lee	Marketing	70000	2018-11-21	<a href="mailto:david.lee@email.com">david.lee@email.com</a>	2
6	Sophia Davis	Sales	48000	2021-02-28	<a href="mailto:sophia.davis@email.com">sophia.davis@email.com</a>	1
7	Liam Walker	IT	75000	2022-05-12	<a href="mailto:liam.walker@email.com">liam.walker@email.com</a>	4
8	Olivia Martinez	HR	68000	2020-06-10	<a href="mailto:olivia.martinez@email.com">olivia.martinez@email.com</a>	4
9	Noah Wilson	Sales	59000	2021-09-01	<a href="mailto:noah.wilson@email.com">noah.wilson@email.com</a>	3
10	Isabella Lee	Marketing	80000	2017-03-01	<a href="mailto:isabella.lee@email.com">isabella.lee@email.com</a>	2

## Day 13 Tasks:

### 1. Retrieving Data:

- a. Retrieve all data from the employee table.

### 2. Selecting Specific Columns:

- a. Retrieve only the name and salary of all employees.

### 3. Using DISTINCT:

- a. Retrieve a list of unique departments from the employee table.

### 4. Filtering Using the WHERE Clause:

- a. Retrieve employees whose salary is greater than 55,000.

### 5. Using Comparison Operators:

- a. Retrieve employees whose salary is less than or equal to 50,000.

### 6. Using Logical Operators (AND, OR):

- a. Retrieve employees working in the 'Sales' department and earning more than 45,000.

### 7. Using NOT Logical Operator:

- a. Retrieve employees who are not working in the 'HR' department.

### 8. Pattern Matching with LIKE:

- a. Retrieve employees whose email address contains 'email.com'.

### 9. Using BETWEEN for Range Filtering:

- a. Retrieve employees whose salary is between 50,000 and 70,000.

### 10. Using IN for List Filtering:

- Retrieve employees working in the 'Sales' or 'Marketing' department.

### 11. Filtering NULL Values:

- Retrieve employees who do not have a manager (i.e., manager\_id is NULL).

### 12. Sorting Using ORDER BY:

- Retrieve employee names and salaries, sorted by salary in descending order.

### 13. Sorting by Multiple Columns:

- Retrieve employee names, departments, and salaries, sorted first by department in ascending order, then by salary in descending order.

## Mini Project 1: Employee Salary Analysis

**Objective:** Build a simple project that helps HR managers analyze employee salaries based on various criteria such as department, salary range, and sorting.

### Tasks:

1. **Create a database and a table** called employees with the following columns: id, name, department, salary, joining\_date, and email.
2. **Insert data** into the table for 10 employees across different departments (Sales, Marketing, HR, IT).
3. **Data Retrieval:**
  - a. Retrieve all data from the employees table.
  - b. Retrieve only the name and salary of employees.
  - c. Use DISTINCT to find unique department names in the table.
4. **Data Filtering:**
  - a. Retrieve employees whose salary is greater than 60,000.
  - b. Retrieve employees working in the Sales or Marketing departments.
  - c. Retrieve employees whose email contains the domain "email.com".
  - d. Retrieve employees with a salary between 50,000 and 80,000.
5. **Sorting Data:**
  - a. Sort employees by salary in descending order.
  - b. Sort employees by department in ascending order and salary in descending order.

**Outcome:** A query-driven analysis of employees' details to better understand salary distributions, department-wise data, and sorting based on specific conditions.

## Mini Project 2: Customer Orders Management System

**Objective:** Build a system to manage customer orders in an e-commerce platform, where you can retrieve, filter, and sort order data based on various criteria like order status, customer, and order amount.

### Tasks:

1. **Create a database and a table** called orders with the following columns: order\_id, customer\_id, order\_date, order\_amount, status (Pending, Shipped, Delivered), and shipping\_address.
2. **Insert data** into the orders table for at least 10 different customer orders with varying amounts, statuses, and dates.
3. **Data Retrieval:**
  - a. Retrieve all orders from the orders table.
  - b. Retrieve only the order\_id and order\_amount for all orders.
  - c. Use DISTINCT to list all unique order statuses (Pending, Shipped, Delivered).
4. **Data Filtering:**
  - a. Retrieve orders where the order\_amount is greater than 50.
  - b. Retrieve orders placed by customers with customer\_id 3, 5, and 7.
  - c. Retrieve orders that were placed between '2022-01-01' and '2022-12-31'.
  - d. Retrieve orders where the status is either 'Shipped' or 'Delivered'.
5. **Sorting Data:**
  - a. Sort orders by order\_amount in descending order.
  - b. Sort orders by order\_date in ascending order and order\_amount in descending order.

**Outcome:** A query-driven order management system that allows you to retrieve and filter customer orders based on order amount, status, and other key criteria, providing insights into customer behavior and order trends.

# Day 14

## Aggregating Data

### Aggregate Functions

Aggregate functions are used to perform calculations on multiple rows of data and return a single result.

- **COUNT()**: Returns the number of rows that match a specified condition.

#### Syntax:

```
SELECT COUNT(column_name)
FROM table_name
WHERE condition;
```

**Example:** Count how many employees are in a specific department:

```
SELECT COUNT(*) FROM employees WHERE department = 'Sales';
```

- **SUM()**: Returns the total sum of a numeric column.

#### Syntax:

```
SELECT SUM(column_name)
FROM table_name
WHERE condition;
```

**Example:** Calculate the total salary of all employees:

```
SELECT SUM(salary) FROM employees;
```

- **AVG()**: Returns the average value of a numeric column.

**Syntax:**

```
SELECT AVG(column_name)  
FROM table_name;
```

**Example:** Find the average salary of all employees:

```
SELECT AVG(salary) FROM employees;
```

- **MIN()**: Returns the minimum value of a column.

**Syntax:**

```
SELECT MIN(column_name)  
FROM table_name;
```

**Example:** Find the minimum price of a product:

```
SELECT MIN(price) FROM products;
```

- **MAX()**: Returns the maximum value of a column.

**Syntax:**

```
SELECT MAX(column_name)  
FROM table_name;
```

**Example:** Find the maximum order amount from the orders table:

```
SELECT MAX(order_amount) FROM orders;
```

## Grouping Results with GROUP BY

GROUP BY is used to group rows that have the same values into summary rows, like finding the total amount of sales per department.

### Syntax:

```
SELECT column_name, AGGREGATE_FUNCTION(column_name)
FROM table_name
GROUP BY column_name;
```

**Example:** Get the total salary for each department:

```
SELECT department, SUM(salary)
FROM employees
GROUP BY department;
```

## Filtering Groups with HAVING

The HAVING clause is used to filter groups created by the GROUP BY clause. It works like the WHERE clause, but for grouped data.

### Syntax:

```
SELECT column_name, AGGREGATE_FUNCTION(column_name)
FROM table_name
GROUP BY column_name
HAVING condition;
```

**Example:** Get the departments where the total salary is greater than 100,000:

```
SELECT department, SUM(salary)
FROM employees
GROUP BY department
HAVING SUM(salary) > 100000;
```

### Distinguishing Between WHERE and HAVING

- **WHERE** is used to filter rows before the grouping operation.
- **HAVING** is used to filter groups after the GROUP BY operation.

#### Example:

- Use WHERE to filter individual rows:

```
SELECT department, salary FROM employees WHERE salary > 50000;
```

- Use HAVING to filter grouped results:

```
SELECT department, SUM(salary) FROM employees GROUP BY department
HAVING SUM(salary) > 100000;
```

## Joins

Joins are used to combine rows from two or more tables based on a related column.

### INNER JOIN

An INNER JOIN returns rows when there is a match in both tables.

**Syntax:**

```
SELECT columns  
FROM table1  
INNER JOIN table2  
ON table1.common_column = table2.common_column;
```

**Example:** Get the orders and customer details:

```
SELECT orders.order_id, customers.name  
FROM orders  
INNER JOIN customers  
ON orders.customer_id = customers.customer_id;
```

**LEFT JOIN (or LEFT OUTER JOIN)**

A LEFT JOIN returns all rows from the left table, and the matching rows from the right table. If no match is found, NULL is returned for columns from the right table.

**Syntax:**

```
SELECT columns  
FROM table1  
LEFT JOIN table2  
ON table1.common_column = table2.common_column;
```

**Example:** Get all employees and their department names, even if they don't belong to any department:

```
SELECT employees.name, departments.department_name  
FROM employees  
LEFT JOIN departments  
ON employees.department_id = departments.department_id;
```

### RIGHT JOIN (or RIGHT OUTER JOIN)

A RIGHT JOIN returns all rows from the right table, and the matching rows from the left table. If no match is found, NULL is returned for columns from the left table.

#### Syntax:

```
SELECT columns  
FROM table1  
RIGHT JOIN table2  
ON table1.common_column = table2.common_column;
```

**Example:** Get all products and their suppliers, including suppliers with no products:

```
SELECT products.product_name, suppliers.supplier_name  
FROM products  
RIGHT JOIN suppliers  
ON products.supplier_id = suppliers.supplier_id;
```

### FULL OUTER JOIN

A FULL OUTER JOIN returns all rows from both tables, with NULL where there is no match.

**Syntax:**

```
SELECT columns
FROM table1
FULL OUTER JOIN table2
ON table1.common_column = table2.common_column;
```

**Example:** Get all customers and their orders, including customers with no orders and orders with no associated customers:

```
SELECT customers.name, orders.order_id
FROM customers
FULL OUTER JOIN orders
ON customers.customer_id = orders.customer_id;
```

**Self Joins**

A self join is a regular join, but the table is joined with itself.

What is a Self Join?

A self join is useful when you want to compare rows within the same table.

Using Aliases for Self Joins

When performing a self join, you use aliases to differentiate between the two instances of the same table.

**Syntax:**

```
SELECT a.column_name, b.column_name
FROM table_name a, table_name b
WHERE a.common_column = b.common_column;
```

**Example:** Get a list of employees and their managers (assuming employee\_id and manager\_id are in the same employees table):

```
SELECT e.name AS Employee, m.name AS Manager
FROM employees e
LEFT JOIN employees m
ON e.manager_id = m.employee_id;
```

This self join matches each employee to their manager by using aliases for the employees table.

## Mini Project 1: Student Performance Analysis

**Description:** In this project, you'll work with a database of student performance in various subjects and analyze the data using aggregate functions and joins.

### 1. Database Setup

- Create a students table with the following columns:
  - student\_id (Primary Key)
  - student\_name
  - email
  - enrollment\_date
- Create a courses table with the following columns:
  - course\_id (Primary Key)
  - course\_name
- Create a grades table with the following columns:
  - grade\_id (Primary Key)
  - student\_id (Foreign Key to students)
  - course\_id (Foreign Key to courses)
  - grade (Numeric value representing the grade, e.g., 85, 90)

## 2. Insert Sample Data

Insert sample data into the students, courses, and grades tables.

### Students Table Sample Data:

student_id	student_name	email	enrollment_date
1	Alice	<a href="mailto:alice@example.com">alice@example.com</a>	2020-01-15
2	Bob	<a href="mailto:bob@example.com">bob@example.com</a>	2020-02-20
3	Charlie	<a href="mailto:charlie@example.com">charlie@example.com</a>	2020-03-25
4	Dave	<a href="mailto:dave@example.com">dave@example.com</a>	2020-04-30
5	Eve	<a href="mailto:eve@example.com">eve@example.com</a>	2020-05-05

### Courses Table Sample Data:

course_id	course_name
1	Math
2	English
3	Science
4	History

### Grades Table Sample Data:

grade_id	student_id	course_id	grade
1	1	1	85
2	1	2	90
3	2	1	70
4	2	3	80
5	3	2	95
6	3	4	88
7	4	1	92
8	4	3	85

9	5	2	78
10	5	4	91

### 3. Tasks

1. **Task 1:** Find the total number of students enrolled in the system.
  - a. **Query:** `SELECT COUNT(*) AS total_students  
FROM students;`
2. **Task 2:** Calculate the average grade for each student.
  - a. **Query:** `SELECT student_id, AVG(grade) AS avg_grade  
FROM grades  
GROUP BY student_id;`
3. **Task 3:** Find the highest grade in each course.
  - a. **Query:** `SELECT course_id, MAX(grade) AS max_grade  
FROM grades  
GROUP BY course_id;`
4. **Task 4:** Calculate the total number of students in each course.
  - a. **Query:** `SELECT course_id, COUNT(student_id) AS student_count  
FROM grades  
GROUP BY course_id;`
5. **Task 5:** Retrieve students who have an average grade greater than 85.
  - a. **Query:** `SELECT student_id, AVG(grade) AS avg_grade  
FROM grades  
GROUP BY student_id  
HAVING AVG(grade) > 85;`

6. **Task 6:** List the courses with an average grade below 80.
- Query:** SELECT course\_id, AVG(grade) AS avg\_course\_grade  
FROM grades  
GROUP BY course\_id  
HAVING AVG(grade) < 80;
7. **Task 7:** Retrieve the names of students who scored more than 90 in Math.
- Query:** SELECT s.student\_name  
FROM students s  
JOIN grades g ON s.student\_id = g.student\_id  
WHERE g.course\_id = 1 AND g.grade > 90;
8. **Task 8:** Find students who have taken both "Math" and "English".
- Query:** SELECT s.student\_name  
FROM students s  
JOIN grades g1 ON s.student\_id = g1.student\_id  
JOIN grades g2 ON s.student\_id = g2.student\_id  
WHERE g1.course\_id = 1 AND g2.course\_id = 2;
9. **Task 9:** Retrieve the course name and the average grade for each course using INNER JOIN.
- Query:** SELECT c.course\_name, AVG(g.grade) AS avg\_grade  
FROM courses c  
INNER JOIN grades g ON c.course\_id = g.course\_id  
GROUP BY c.course\_name;
10. **Task 10:** Get the list of students and the courses they have taken, using LEFT JOIN.
- Query:** SELECT s.student\_name, c.course\_name  
FROM students s  
LEFT JOIN grades g ON s.student\_id = g.student\_id

```
LEFT JOIN courses c ON g.course_id = c.course_id;
```

**11. Task 11:** Use a FULL OUTER JOIN to list all students and courses, even if no courses have been assigned to a student.

- **Query:**

```
SELECT s.student_name, c.course_name
FROM students s
FULL OUTER JOIN grades g ON s.student_id = g.student_id
FULL OUTER JOIN courses c ON g.course_id = c.course_id;
```

**12. Task 12:** Find the student who has the highest average grade across all courses.

- **Query:**

```
SELECT student_id, AVG(grade) AS avg_grade
FROM grades
GROUP BY student_id
ORDER BY avg_grade DESC
LIMIT 1;
```

**13. Task 13:** Perform a self-join to find students who scored the same grade in both "Math" and "Science".

- **Query:**

```
SELECT s1.student_name AS student1, s2.student_name AS student2
FROM grades g1
JOIN grades g2 ON g1.student_id = g2.student_id
JOIN students s1 ON g1.student_id = s1.student_id
JOIN students s2 ON g2.student_id = s2.student_id
WHERE g1.course_id = 1 AND g2.course_id = 3 AND g1.grade = g2.grade;
```

## Summary of the Project

This mini-project gives you practical exposure to:

- Using aggregate functions (COUNT(), AVG(), MAX(), etc.) for analyzing student data.
- Applying GROUP BY and HAVING to group results and filter aggregated data.
- Using different types of joins (INNER JOIN, LEFT JOIN, FULL OUTER JOIN) to connect student, course, and grades data.
- Demonstrating a self-join to compare students' performance within the same dataset.

This project will help you build the skills to analyze educational or any other type of structured data effectively using SQL.

## Mini Project 2: Sales Order Database

**Description:** In this project, you will work with a sales order database to calculate aggregate metrics and perform joins to analyze sales data.

### Step-by-Step Solution:

#### 1. Database Setup

- Create an orders table with the following columns:
  - order\_id (Primary Key)
  - customer\_id
  - order\_date
  - total\_amount
- Create a customers table with the following columns:
  - customer\_id (Primary Key)
  - customer\_name

- customer\_email
- Create an order\_items table with the following columns:
  - order\_item\_id (Primary Key)
  - order\_id (Foreign Key to orders)
  - product\_name
  - quantity
  - price

## 2. Insert Sample Data

Insert sample data into the tables.

### Orders Table Sample Data:

order_id	customer_id	order_date	total_amount
1	101	2021-05-10	250
2	102	2021-06-15	450
3	103	2021-07-20	700
4	101	2021-08-25	300

### Customers Table Sample Data:

customer_id	customer_name	customer_email
101	Alice	<a href="mailto:alice@example.com">alice@example.com</a>
102	Bob	<a href="mailto:bob@example.com">bob@example.com</a>
103	Charlie	<a href="mailto:charlie@example.com">charlie@example.com</a>

### Order Items Table Sample Data:

order_item_id	order_id	product_name	quantity	price
1	1	Laptop	1	150
2	1	Mouse	1	50
3	2	Keyboard	1	200
4	2	Headset	1	250
5	3	Laptop	1	500
6	4	Mouse	2	50

### 3. Tasks

1. **Task 1:** Find the total number of orders.

a. **Query:** `SELECT COUNT(*) AS total_orders  
FROM orders;`

2. **Task 2:** Calculate the average order amount.

a. **Query:** `SELECT AVG(total_amount) AS avg_order_amount  
FROM orders;`

3. **Task 3:** Find the highest total order amount.

a. **Query:** `SELECT MAX(total_amount) AS max_order_amount  
FROM orders;`

4. **Task 4:** Get the total sales amount for each customer.

a. **Query:** `SELECT customer_id, SUM(total_amount) AS total_sales  
FROM orders  
GROUP BY customer_id;`

5. **Task 5:** List customers who have total sales above 400.

a. **Query:** `SELECT customer_id, SUM(total_amount) AS total_sales  
FROM orders  
GROUP BY customer_id`

HAVING SUM(total\_amount) > 400;

6. **Task 6:** Get the total quantity of each product ordered.

a. **Query:** SELECT product\_name, SUM(quantity) AS total\_quantity  
FROM order\_items  
GROUP BY product\_name;

7. **Task 7:** Find the total number of orders made by each customer.

a. **Query:** SELECT customer\_id, COUNT(\*) AS order\_count  
FROM orders  
GROUP BY customer\_id;

8. **Task 8:** Retrieve customer details along with their order amounts using INNER JOIN.

a. **Query:** SELECT c.customer\_name, c.customer\_email, o.total

\_amount FROM customers c INNER JOIN orders o ON c.customer\_id =  
o.customer\_id; ``

9. **Task 9:** Retrieve all orders with their product names and quantities using LEFT JOIN.

a. **Query:** SELECT o.order\_id, oi.product\_name, oi.quantity  
FROM orders o  
LEFT JOIN order\_items oi  
ON o.order\_id = oi.order\_id;

10. **Task 10:** Use a self-join to find orders where the total amount is equal to the total amount in another order by the same customer.

• **Query:** SELECT o1.order\_id AS order1, o2.order\_id AS order2  
FROM orders o1

```

INNER JOIN orders o2
ON o1.customer_id = o2.customer_id
WHERE o1.total_amount = o2.total_amount AND o1.order_id != o2.order_id;

```

**Dataset:**

## 1. students Table:

student_id	student_name	course_id	grade	enrollment_date
1	John	101	A	2021-08-15
2	Alice	102	B	2020-06-20
3	Bob	101	A	2022-01-10
4	Carol	103	C	2021-03-18
5	Dave	104	B	2020-11-05
6	Eve	102	A	2022-02-23
7	Frank	104	C	2021-10-12
8	Grace	103	B	2022-04-08
9	Hannah	101	A	2020-09-15
10	Ian	104	A	2021-06-30

## 2. courses Table:

course_id	course_name	department
101	Computer Science	Engineering
102	Mathematics	Science

103	Physics	Science
104	Mechanical Engineering	Engineering

**Day 14 Tasks:**

1. Find the total number of students enrolled in the courses.
2. Calculate the total number of students in each course.
3. Calculate the average grade of students for each course (treat grades numerically, i.e., A=4, B=3, C=2).
4. Find the student with the lowest grade in each course.
5. Find the courses with more than 2 students enrolled.
6. Get the courses where the average grade is greater than 3.
7. Retrieve the list of students along with the name of the course they are enrolled in.
8. Find all students and their courses, including students not enrolled in any course.
9. Retrieve a list of students and courses, where the student has enrolled in at least one course.
10. List all courses with or without students enrolled.
11. Retrieve a list of students and courses, including courses without students enrolled.
12. Find students who are enrolled in the same course as another student.
13. List students who have the same grade as another student in the same course.

## Mini Project 1: Sales Analysis

Dataset 1: sales Table

sale_id	product_id	sale_date	amount
1	101	2021-05-01	500
2	102	2021-05-02	1500
3	103	2021-06-15	300
4	101	2021-06-16	700
5	104	2021-07-05	1200
6	102	2021-08-21	800
7	103	2021-09-03	450
8	104	2021-09-20	200
9	101	2021-10-01	550
10	102	2021-11-13	1000

Dataset 2: products Table

product_id	product_name	category
101	Laptop	Electronics
102	Smartphone	Electronics
103	Tablet	Electronics
104	Headphones	Accessories

### Mini Project Tasks 1:

#### 1. Using Aggregate Functions:

- a. Find the total sales amount for each product.
- b. Find the highest sale amount for each product.
- c. Calculate the average sale amount for each product category.

**2. Grouping Results with GROUP BY:**

- a. Group the sales by product category and find the total sales for each category.

**3. Filtering Groups with HAVING:**

- a. Display the product categories where the total sales amount is greater than 2000.

**4. Joins:**

- a. Use an INNER JOIN to list all products along with their sales amount.
- b. Use a LEFT JOIN to list all products, even those with no sales, along with their sales amount.

**5. Self Joins:**

- a. Find products that have sales in the same month as another product.

**Mini Project 2: Employee and Department Management**

Dataset 1: employees Table

<b>emp_id</b>	<b>emp_name</b>	<b>dept_id</b>	<b>salary</b>
1	John	101	5000
2	Alice	102	4000
3	Bob	101	5500
4	Carol	103	6000
5	Dave	102	4500
6	Eve	103	7000
7	Frank	101	4500
8	Grace	104	5500
9	Hannah	104	5000
10	Ian	102	4700

## Dataset 2: departments Table

dept_id	dept_name	location
101	HR	New York
102	Finance	Chicago
103	Marketing	San Francisco
104	IT	Austin

## Mini Project Tasks 2:

### 1. Using Aggregate Functions:

- a. Calculate the total salary expense in each department.
- b. Find the employee with the highest salary in each department.
- c. Calculate the average salary for each department.

### 2. Grouping Results with GROUP BY:

- a. Group employees by department and find the total salary paid to each department.

### 3. Filtering Groups with HAVING:

- a. Display departments with an average salary greater than 5000.

### 4. Joins:

- a. Use an INNER JOIN to list all employees along with their department names.
- b. Use a LEFT JOIN to list all employees, including those without a department, along with their department name.

### 5. Self Joins:

- a. Find employees who have the same salary as other employees in the same department.

# Day 15

## Subqueries and Advanced Queries in SQL

### 1. Subqueries

#### What is a Subquery?

A subquery is a query nested inside another SQL query. It is used to retrieve data that will be used by the main (outer) query.

#### When to Use Subqueries?

- To filter data dynamically within a WHERE clause.
- To return a calculated value for comparison.
- To retrieve aggregated data (e.g., getting the highest salary of an employee).
- To replace complex joins in some cases.

#### Types of Subqueries

1. **Non-correlated subqueries:** Independent of the outer query and executes first.
2. **Correlated subqueries:** Dependent on the outer query, executing row by row.

#### Using Subqueries in SELECT, FROM, and WHERE Clauses

##### 1. Subquery in SELECT Clause

Used to return a calculated value.

Example: Find employees and their salaries compared to the highest salary in the company.

```
SELECT emp_name, salary,  
       (SELECT MAX(salary) FROM employees) AS highest_salary  
FROM employees;
```

## 2. Subquery in FROM Clause

A subquery is treated as a temporary table.

Example: Get departments with an average salary greater than 5000.

```
SELECT dept_id, avg_salary  
FROM (SELECT dept_id, AVG(salary) AS avg_salary FROM employees GROUP BY  
dept_id) AS dept_avg  
WHERE avg_salary > 5000;
```

## 3. Subquery in WHERE Clause

Used for filtering results.

Example: Find employees earning more than the average salary.

```
SELECT emp_name, salary  
FROM employees  
WHERE salary > (SELECT AVG(salary) FROM employees);
```

## Correlated vs. Non-Correlated Subqueries

Feature	Non-Correlated Subquery	Correlated Subquery
Execution	Runs once before outer query	Runs once per row of the outer query
Dependency	Independent of the outer query	Depends on the outer query's current row
Performance	Faster (precomputed)	Slower (executed multiple times)

### Example of a Correlated Subquery

Find employees whose salary is above the average salary in their department.

```
SELECT emp_name, salary, dept_id
FROM employees e1
WHERE salary > (SELECT AVG(salary) FROM employees e2 WHERE e1.dept_id =
e2.dept_id);
```

## 2. UNION and INTERSECT Operations

### Using UNION to Combine Results from Multiple Queries

- The UNION operator combines results from two or more queries and removes duplicates.

#### Syntax:

```
SELECT column1 FROM table1
```

```
UNION
```

```
SELECT column1 FROM table2;
```

Example: Retrieve a combined list of product names from the electronics and furniture tables.

```
SELECT product_name FROM electronics  
UNION  
SELECT product_name FROM furniture;
```

### Difference Between UNION and UNION ALL

- UNION removes duplicate records.
- UNION ALL keeps duplicates for better performance.

#### Example:

```
SELECT product_name FROM electronics  
UNION ALL  
SELECT product_name FROM furniture;
```

### Using INTERSECT to Get Common Results

- The INTERSECT operator returns only matching rows from two queries.

Example: Find employees who work both in IT and Finance departments.

```
SELECT emp_name FROM employees WHERE dept_id = 101  
INTERSECT  
SELECT emp_name FROM employees WHERE dept_id = 102;
```

### Using EXCEPT (or MINUS) to Get Unique Results

- EXCEPT (or MINUS in some databases) removes records present in the second query.

Example: Find employees in IT department but not in HR department.

```
SELECT emp_name FROM employees WHERE dept_id = 104
EXCEPT
SELECT emp_name FROM employees WHERE dept_id = 101;
```

### **3. Complex Queries**

#### **Combining JOIN, GROUP BY, and Aggregate Functions**

- We can join multiple tables, group results, and apply aggregate functions.

Example: Find the total salary paid in each department along with the department name.

```
SELECT d.dept_name, SUM(e.salary) AS total_salary
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id
GROUP BY d.dept_name;
```

#### **Case Statements and Conditional Aggregation (CASE WHEN)**

- The CASE statement allows applying conditional logic.

Example: Classify employees based on their salary.

```
SELECT emp_name, salary,
CASE
    WHEN salary > 6000 THEN 'High'
    WHEN salary BETWEEN 4000 AND 6000 THEN 'Medium'
    ELSE 'Low'
END AS salary_category
FROM employees;
```

## Working with Dates and Times

- SQL provides **date functions** like CURRENT\_DATE, DATE\_ADD, DATE\_SUB.

**Example:** Get employees who joined within the last 6 months.

```
SELECT emp_name, hire_date
FROM employees
WHERE hire_date >= DATE_SUB(CURRENT_DATE, INTERVAL 6 MONTH);
```

## Conclusion

- Subqueries allow nested querying and comparisons.
- UNION, INTERSECT, and EXCEPT are useful for combining query results.
- Complex queries use JOIN, GROUP BY, and conditional statements like CASE WHEN for better insights.
- Date functions help analyze time-based data.

## Mini Project 1: Employee Performance Analysis

### Objective:

Analyze employee performance based on salaries, departments, and hire dates using subqueries, UNION, INTERSECT, and complex queries.

### Dataset: employees and departments Tables

```
CREATE TABLE departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(50)
);
```

```

INSERT INTO departments (dept_id, dept_name) VALUES
(1, 'IT'),
(2, 'Finance'),
(3, 'HR'),
(4, 'Sales');

CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50),
    salary DECIMAL(10,2),
    hire_date DATE,
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES departments(dept_id)
);

```

```

INSERT INTO employees (emp_id, emp_name, salary, hire_date, dept_id) VALUES
(101, 'Alice', 7000, '2018-05-20', 1),
(102, 'Bob', 5000, '2019-07-15', 2),
(103, 'Charlie', 6000, '2020-02-10', 1),
(104, 'David', 5500, '2021-06-01', 3),
(105, 'Eva', 8000, '2017-09-23', 4),
(106, 'Frank', 4800, '2022-01-18', 2),
(107, 'Grace', 6200, '2019-12-30', 1);

```

## Tasks & Solutions

- Find employees who earn more than the average salary of all employees.

```

SELECT emp_name, salary
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);

```

**Explanation:** Uses a subquery in WHERE clause to filter employees earning more than the average salary.

2. Find departments with employees having a salary greater than 6000.

```
SELECT dept_name
FROM departments
WHERE dept_id IN (
    SELECT dept_id FROM employees WHERE salary > 6000
);
```

**Explanation:** Uses a subquery to filter departments based on employees earning above 6000.

3. List employees who joined within the last 3 years.

```
SELECT emp_name, hire_date
FROM employees
WHERE hire_date >= DATE_SUB(CURRENT_DATE, INTERVAL 3 YEAR);
```

**Explanation:** Uses DATE\_SUB to find employees hired within the last 3 years.

4. Use UNION to combine employees from IT and Finance departments.

```
SELECT emp_name FROM employees WHERE dept_id = 1
UNION
SELECT emp_name FROM employees WHERE dept_id = 2;
```

**Explanation:** Combines employees from IT and Finance departments, removing duplicates.

5. Use INTERSECT to find employees present in both IT and HR departments.

```
SELECT emp_name FROM employees WHERE dept_id = 1
```

```
INTERSECT
```

```
SELECT emp_name FROM employees WHERE dept_id = 3;
```

**Explanation:** Retrieves employees working in both IT and HR.

6. Categorize employees as 'Senior' or 'Junior' based on salary.

```
SELECT emp_name, salary,
```

```
CASE
```

```
WHEN salary > 6000 THEN 'Senior'
```

```
ELSE 'Junior'
```

```
END AS category
```

```
FROM employees;
```

**Explanation:** Uses a CASE statement to classify employees based on salary.

## Mini Project 2: Product Sales Analysis

### Objective:

Analyze product sales performance using subqueries, joins, aggregation, and UNION operations.

### Dataset: products, sales, and customers Tables

```
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(50),
    category VARCHAR(50),
    price DECIMAL(10,2)
```

```
);
```

```
INSERT INTO products (product_id, product_name, category, price) VALUES  
(1, 'Laptop', 'Electronics', 1000),  
(2, 'Phone', 'Electronics', 700),  
(3, 'Table', 'Furniture', 300),  
(4, 'Chair', 'Furniture', 150),  
(5, 'Headphones', 'Electronics', 100);
```

```
CREATE TABLE customers (  
    customer_id INT PRIMARY KEY,  
    customer_name VARCHAR(50),  
    city VARCHAR(50)  
);
```

```
INSERT INTO customers (customer_id, customer_name, city) VALUES  
(1, 'John', 'New York'),  
(2, 'Sarah', 'Los Angeles'),  
(3, 'Mike', 'Chicago'),  
(4, 'Emma', 'Houston');
```

```
CREATE TABLE sales (  
    sale_id INT PRIMARY KEY,  
    product_id INT,  
    customer_id INT,  
    quantity INT,  
    sale_date DATE,  
    FOREIGN KEY (product_id) REFERENCES products(product_id),  
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)  
);
```

```
INSERT INTO sales (sale_id, product_id, customer_id, quantity, sale_date) VALUES  
(101, 1, 1, 2, '2023-01-15'),  
(102, 2, 2, 1, '2023-02-10'),  
(103, 3, 3, 4, '2023-03-05'),  
(104, 4, 4, 3, '2023-03-20'),  
(105, 1, 2, 1, '2023-04-12'),  
(106, 5, 3, 5, '2023-05-06');
```

### Tasks & Solutions

1. Find the most expensive product.

```
SELECT product_name, price  
FROM products  
WHERE price = (SELECT MAX(price) FROM products);
```

**Explanation:** Uses a subquery in WHERE clause to get the highest-priced product.

2. Get total quantity sold per product category.

```
SELECT p.category, SUM(s.quantity) AS total_sold  
FROM sales s  
JOIN products p ON s.product_id = p.product_id  
GROUP BY p.category;
```

**Explanation:** Uses JOIN and GROUP BY to sum quantity sold per category.

3. List customers who have purchased both 'Laptop' and 'Phone'.

```
SELECT customer_name FROM customers  
WHERE customer_id IN (  
    SELECT customer_id FROM sales WHERE product_id = 1  
    AND product_id = 2)
```

```
INTERSECT
SELECT customer_id FROM sales WHERE product_id = 2
);
```

**Explanation:** Uses INTERSECT to find customers who bought both Laptop and Phone.

4. Find customers who made purchases but are not from New York.

```
SELECT customer_name FROM customers
WHERE customer_id IN (SELECT customer_id FROM sales)
EXCEPT
SELECT customer_name FROM customers WHERE city = 'New York';
```

**Explanation:** Uses EXCEPT to remove New York customers.

5. Find the most sold product.

```
SELECT product_name
FROM products
WHERE product_id = (
    SELECT product_id FROM sales
    GROUP BY product_id
    ORDER BY SUM(quantity) DESC
    LIMIT 1
);
```

**Explanation:** Uses a subquery with GROUP BY and ORDER BY to get the most sold product.

6. Calculate total sales per product, displaying 'High Sales' for products sold more than 5 units.

```
SELECT p.product_name, SUM(s.quantity) AS total_quantity,
CASE
    WHEN SUM(s.quantity) > 5 THEN 'High Sales'
    ELSE 'Low Sales'
END AS sales_category
FROM sales s
JOIN products p ON s.product_id = p.product_id
GROUP BY p.product_name;
```

**Explanation:** Uses JOIN, GROUP BY, and CASE WHEN for classification.

## Datasets :

### Dataset: Online Store Database

This dataset contains **Customers, Orders, Products, and Sales details**.

#### 1. customers Table

```
CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    customer_name VARCHAR(100),
    city VARCHAR(50),
    join_date DATE
);
```

```
INSERT INTO customers (customer_id, customer_name, city, join_date) VALUES
(1, 'John Doe', 'New York', '2020-01-15'),
```

```
(2, 'Sarah Smith', 'Los Angeles', '2021-06-22'),  
(3, 'Michael Brown', 'Chicago', '2019-11-10'),  
(4, 'Emma Davis', 'Houston', '2022-05-05'),  
(5, 'Robert Wilson', 'San Francisco', '2023-02-14');
```

## 2. products Table

```
CREATE TABLE products (  
    product_id INT PRIMARY KEY,  
    product_name VARCHAR(100),  
    category VARCHAR(50),  
    price DECIMAL(10,2)  
);
```

```
INSERT INTO products (product_id, product_name, category, price) VALUES  
(101, 'Laptop', 'Electronics', 1200.00),  
(102, 'Smartphone', 'Electronics', 800.00),  
(103, 'Tablet', 'Electronics', 500.00),  
(104, 'Office Chair', 'Furniture', 200.00),  
(105, 'Desk', 'Furniture', 400.00);
```

## 3. orders Table

```
CREATE TABLE orders (  
    order_id INT PRIMARY KEY,  
    customer_id INT,  
    order_date DATE,  
    total_amount DECIMAL(10,2),  
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)  
);
```

```
INSERT INTO orders (order_id, customer_id, order_date, total_amount) VALUES
```

```
(1001, 1, '2023-03-10', 1300.00),
(1002, 2, '2023-04-15', 800.00),
(1003, 3, '2023-05-20', 600.00),
(1004, 4, '2023-06-25', 700.00),
(1005, 5, '2023-07-30', 1400.00);
```

#### 4. order\_details Table

```
CREATE TABLE order_details (
    order_detail_id INT PRIMARY KEY,
    order_id INT,
    product_id INT,
    quantity INT,
    FOREIGN KEY (order_id) REFERENCES orders(order_id),
    FOREIGN KEY (product_id) REFERENCES products(product_id)
);
```

```
INSERT INTO order_details (order_detail_id, order_id, product_id, quantity)
VALUES
(1, 1001, 101, 1),
(2, 1001, 103, 2),
(3, 1002, 102, 1),
(4, 1003, 105, 1),
(5, 1004, 104, 2),
(6, 1005, 101, 1),
(7, 1005, 105, 2);
```

## Day 15 Tasks :

1. Retrieve customers who placed an order with a total amount greater than the average order amount.
2. Find the most expensive product ordered.
3. List all customers who have not placed any orders.
4. Get all orders where the total amount is greater than the average order total.
5. Get the latest order placed by each customer.
6. Show orders where at least one product belongs to the 'Electronics' category.
7. Show customers who have purchased both an 'Electronics' and a 'Furniture' product.
8. Display customers who have placed an order but are not from New York.
9. Retrieve the number of products ordered per category.
10. List the total sales per product, classifying them as 'High Sales' if sold more than 2 units.
11. Display the total sales revenue per city.
12. Retrieve customers who have placed at least two different orders.
13. Find customers who joined more than 2 years ago and have placed an order.

## Mini Project 1: Customer Insights and Order Analysis

### Dataset:

Customers Table

customer_id	name	city	signup_date
1	John Doe	New York	2020-01-15
2	Jane Smith	Los Angeles	2019-06-22
3	Mike Johnson	Chicago	2021-03-10
4	Emily Davis	Houston	2022-07-05

Orders Table

order_id	customer_id	order_date	total_amount
101	1	2023-02-10	250
102	2	2023-05-15	400
103	3	2023-06-20	100
104	1	2023-08-05	150
105	4	2023-09-01	300

### Project Tasks:

1. Retrieve customers who have placed an order with a total amount greater than the average order amount.
2. Find the most recent order placed by each customer.
3. List customers who have never placed an order using a subquery.
4. Show customers who signed up before 2021 and placed an order in 2023.
5. Use UNION to display all customer names along with customers who placed an order.

6. Find customers who placed orders but are not from New York (using EXCEPT or MINUS).
7. Use CASE WHEN to categorize total amounts as 'Low', 'Medium', or 'High' sales.
8. Retrieve the total revenue per city, grouped by city.

## Mini Project 2: Product Sales Performance Analysis

### Dataset:

Products Table

product_id	product_name	category	price
1	Laptop	Electronics	1200
2	Headphones	Electronics	150
3	Sofa	Furniture	800
4	Chair	Furniture	100

Sales Table

sale_id	product_id	sale_date	quantity	total_price
201	1	2023-02-15	2	2400
202	2	2023-03-20	5	750
203	3	2023-05-25	1	800
204	4	2023-06-30	4	400
205	1	2023-07-10	1	1200

## Project Tasks:

1. Find the total quantity sold for each product using GROUP BY.
2. Retrieve the most sold product by using a subquery.
3. List all products that were never sold using a subquery.
4. Use UNION to show all products along with products that have been sold.
5. Use CASE WHEN to classify products based on total sales as ‘Low Sales’, ‘Medium Sales’, and ‘High Sales’.
6. Retrieve the total sales revenue for each category.
7. Find products that have been sold more than the average quantity sold.
8. Retrieve the first sale date for each product using a correlated subquery.

## Day 16

### Data Modifications in SQL

In SQL, data modification operations allow us to insert, update, and delete records from database tables. Additionally, constraints ensure data integrity, and transactions help manage multiple operations efficiently.

#### 1. Inserting Data into Tables (INSERT INTO)

##### Definition:

The INSERT INTO statement is used to add new rows to a table. It can insert data into all columns or specific columns.

**Syntax:**

-- Insert into all columns

```
INSERT INTO table_name VALUES (value1, value2, ...);
```

-- Insert into specific columns

```
INSERT INTO table_name (column1, column2) VALUES (value1, value2);
```

**Example:**

```
CREATE TABLE Employees (
    emp_id INT PRIMARY KEY,
    name VARCHAR(50),
    department VARCHAR(50),
    salary DECIMAL(10,2)
);
```

-- Insert data into all columns

```
INSERT INTO Employees VALUES (1, 'John Doe', 'HR', 50000);
```

-- Insert data into specific columns

```
INSERT INTO Employees (emp_id, name) VALUES (2, 'Jane Smith');
```

## 2. Updating Data in a Table (UPDATE with SET)

### Definition:

The UPDATE statement modifies existing records in a table. The SET clause specifies which columns to update, and the WHERE clause determines which rows are affected.

### Syntax:

```
UPDATE table_name  
SET column1 = value1, column2 = value2  
WHERE condition;
```

### Example:

```
-- Increase salary for employees in HR department  
UPDATE Employees  
SET salary = salary + 5000  
WHERE department = 'HR';
```

```
-- Change department for a specific employee  
UPDATE Employees  
SET department = 'Finance'  
WHERE emp_id = 2;
```

### 3. Deleting Data from a Table (DELETE)

#### Definition:

The DELETE statement removes rows from a table. The WHERE clause is used to specify which rows to delete.

#### Syntax:

```
DELETE FROM table_name WHERE condition;
```

#### Example:

```
-- Remove an employee with ID 2
```

```
DELETE FROM Employees WHERE emp_id = 2;
```

```
-- Delete all employees in the Finance department
```

```
DELETE FROM Employees WHERE department = 'Finance';
```

```
-- Delete all records (Caution: No WHERE clause deletes everything!)
```

```
DELETE FROM Employees;
```

### 4. Constraints and Data Integrity

#### Definition:

Constraints enforce rules on data to maintain accuracy and integrity.

#### Types of Constraints:

1. **PRIMARY KEY** – Ensures a unique identifier for each row.
2. **FOREIGN KEY** – Ensures referential integrity between tables.

3. **UNIQUE** – Prevents duplicate values in a column.
4. **NOT NULL** – Ensures a column cannot have NULL values.
5. **CHECK** – Defines conditions that values must meet.

### **Creating Tables with Constraints:**

```
CREATE TABLE Departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(50) UNIQUE
);
```

```
CREATE TABLE Employees (
    emp_id INT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    department_id INT,
    salary DECIMAL(10,2) CHECK (salary > 3000),
    FOREIGN KEY (department_id) REFERENCES Departments(dept_id)
);
```

## **5. Modifying and Dropping Constraints**

### **Adding Constraints to an Existing Table:**

```
ALTER TABLE Employees ADD CONSTRAINT chk_salary CHECK (salary > 3000);
```

### **Dropping Constraints:**

```
ALTER TABLE Employees DROP CONSTRAINT chk_salary;
```

## 6. Transactions in SQL

### Definition:

A transaction is a set of SQL statements that must be executed together to maintain database consistency. Transactions follow ACID properties:

- **Atomicity** – All or nothing.
- **Consistency** – Maintains database rules.
- **Isolation** – Prevents interference between transactions.
- **Durability** – Changes persist even after system failure.

### Commands in Transactions:

1. **COMMIT** – Saves all changes made in the transaction.
2. **ROLLBACK** – Reverts changes if an error occurs.
3. **SAVEPOINT** – Creates a checkpoint to roll back to.

### Syntax & Example:

```
START TRANSACTION;
```

```
-- Insert two employees
```

```
INSERT INTO Employees VALUES (3, 'Alice Brown', 'IT', 60000);
```

```
INSERT INTO Employees VALUES (4, 'Bob Green', 'IT', 55000);
```

```
-- If something goes wrong, rollback
```

```
ROLLBACK;
```

```
-- If everything is fine, commit the changes
```

```
COMMIT;
```

**Using SAVEPOINT:**

```
START TRANSACTION;
```

```
INSERT INTO Employees VALUES (5, 'Charlie White', 'HR', 70000);
SAVEPOINT sp1;
```

```
UPDATE Employees SET salary = salary + 5000 WHERE department = 'HR';
SAVEPOINT sp2;
```

-- If needed, rollback to a specific savepoint

```
ROLLBACK TO sp1;
```

-- Finalize changes

```
COMMIT;
```

**Summary Table: SQL Data Modifications**

Operation	Purpose	Example
INSERT	Adds new rows	INSERT INTO Employees VALUES (1, 'John', 'HR', 50000);
UPDATE	Modifies existing data	UPDATE Employees SET salary = salary + 5000 WHERE department = 'HR';
DELETE	Removes data	DELETE FROM Employees WHERE emp_id = 2;
COMMIT	Saves transaction changes	COMMIT;
ROLLBACK	Reverts changes	ROLLBACK;
SAVEPOINT	Sets a checkpoint	SAVEPOINT sp1;

## Mini Project 1: Employee Management System

### Objective:

Create a database to manage employees, departments, and salaries. Implement data modifications (INSERT, UPDATE, DELETE), constraints, and transactions to maintain data integrity.

### Step 1: Create the Database and Tables

```
CREATE DATABASE EmployeeDB;
```

```
USE EmployeeDB;
```

```
CREATE TABLE Departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(50) UNIQUE NOT NULL
);
```

```
CREATE TABLE Employees (
    emp_id INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    department_id INT,
    salary DECIMAL(10,2) CHECK (salary > 3000),
    FOREIGN KEY (department_id) REFERENCES Departments(dept_id) ON DELETE
    SET NULL
);
```

### **Step 2: Insert Sample Data**

```
INSERT INTO Departments VALUES (1, 'HR'), (2, 'IT'), (3, 'Finance');
```

```
INSERT INTO Employees VALUES  
(101, 'Alice Johnson', 1, 50000),  
(102, 'Bob Smith', 2, 60000),  
(103, 'Charlie Brown', 3, 70000);
```

### **Step 3: Update Employee Salary**

```
UPDATE Employees  
SET salary = salary + 5000  
WHERE department_id = 2;
```

### **Step 4: Delete an Employee Record**

```
DELETE FROM Employees WHERE emp_id = 103;
```

### **Step 5: Implement Transactions for Safe Data Modification**

```
START TRANSACTION;
```

```
INSERT INTO Employees VALUES (104, 'David Green', 1, 55000);  
SAVEPOINT sp1;
```

```
UPDATE Employees SET salary = salary + 5000 WHERE department_id = 1;  
SAVEPOINT sp2;
```

```
-- Rollback if needed  
ROLLBACK TO sp1;
```

```
-- Commit changes if everything is fine
```

```
COMMIT;
```

### **Expected Outcome:**

- Employee and department details are stored securely.
- Data integrity is maintained with constraints.
- Transaction management ensures safe modifications.

## **Mini Project 2: Online Store Inventory System**

### **Objective:**

Create a simple inventory management system for an online store that allows inserting, updating, and deleting product records while ensuring data integrity and transactional safety.

### **Step 1: Create the Database and Tables**

```
CREATE DATABASE StoreDB;
```

```
USE StoreDB;
```

```
CREATE TABLE Categories (
    category_id INT PRIMARY KEY,
    category_name VARCHAR(50) UNIQUE NOT NULL
);
```

```
CREATE TABLE Products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(100) NOT NULL,
    category_id INT,
```

```
price DECIMAL(10,2) CHECK (price > 0),  
stock INT CHECK (stock >= 0),  
FOREIGN KEY (category_id) REFERENCES Categories(category_id) ON DELETE  
CASCADE  
);
```

### **Step 2: Insert Sample Data**

```
INSERT INTO Categories VALUES (1, 'Electronics'), (2, 'Clothing'), (3, 'Groceries');
```

```
INSERT INTO Products VALUES  
(1001, 'Laptop', 1, 800.00, 10),  
(1002, 'T-shirt', 2, 20.00, 50),  
(1003, 'Apples', 3, 3.00, 100);
```

### **Step 3: Update Product Prices and Stock**

```
UPDATE Products  
SET price = price * 1.10, stock = stock - 5  
WHERE category_id = 1;
```

### **Step 4: Delete Out-of-Stock Products**

```
DELETE FROM Products WHERE stock = 0;
```

### **Step 5: Implement Transactions for Bulk Updates**

```
START TRANSACTION;
```

```
UPDATE Products SET stock = stock - 10 WHERE category_id = 2;  
SAVEPOINT sp1;
```

```
UPDATE Products SET price = price * 0.90 WHERE category_id = 3;
```

```
SAVEPOINT sp2;  
  
-- Rollback if there's an issue  
ROLLBACK TO sp1;  
  
-- Commit if everything is successful  
COMMIT;
```

### **Expected Outcome:**

- Product and category records are securely managed.
- Constraints prevent invalid data.
- Transactions help ensure safe modifications.

### **Day 16 Tasks :**

#### **1. Inserting Data into Tables**

- Create a table Customers with columns customer\_id, name, and email.
- Insert at least 5 customer records using the INSERT INTO statement.

#### **2. Updating Data in a Table**

- Update the email of a specific customer based on their customer\_id.

#### **3. Deleting Data from a Table**

- Delete a customer record where the customer\_id is 3.

#### **4. Creating a Table with Constraints**

- Create a Products table with the following constraints:

- product\_id as PRIMARY KEY
- product\_name as UNIQUE
- price must be greater than 0 using a CHECK constraint

## 5. Inserting Data with Constraints

- Insert 3 valid product records into the Products table.
- Try inserting a product with a negative price and observe the error.

## 6. Modifying Constraints

- Modify the Products table to add a NOT NULL constraint on product\_name.

## 7. Dropping Constraints

- Remove the UNIQUE constraint on product\_name in the Products table.

## 8. Using Foreign Keys for Data Integrity

- Create an Orders table where customer\_id is a FOREIGN KEY referencing Customers(customer\_id).
- Insert 2 order records with valid customer\_id values.
- Try inserting an order with an invalid customer\_id and observe the error.

## 9. Using Transactions: COMMIT and ROLLBACK

- Start a transaction.
- Insert a new customer.
- Update their email.
- Rollback the transaction before committing.
- Check if the changes were applied.

## 10. Using SAVEPOINT in Transactions

- Start a transaction.
- Insert a product and set a SAVEPOINT.
- Update the product price.
- Rollback to the SAVEPOINT (undo the price change but keep the product insertion).

## 11. Implementing ACID Properties

- Demonstrate Atomicity by inserting multiple orders inside a transaction.
- If any insert fails, roll back all the changes.

## 12. Cascading Delete in Foreign Key

- Modify the Orders table so that when a customer is deleted from Customers, their orders are also deleted (ON DELETE CASCADE).
- Delete a customer and check if their orders are also removed.

## 13. Checking Data Integrity with Constraints

- Create a Payments table with a CHECK constraint ensuring that amount must be greater than 0.
- Insert a valid and an invalid payment record to test the constraint.

## **Mini Project 1 (Updated): Student Enrollment System (Using Data Modifications, Constraints, and Transactions)**

### **Project Overview:**

Develop a Student Enrollment System using SQL that allows inserting, updating, and deleting student and course records while ensuring data integrity using constraints and transactions.

### **Database Schema:**

```

CREATE TABLE Courses (
    course_id INT PRIMARY KEY,
    course_name VARCHAR(100) UNIQUE NOT NULL,
    duration INT CHECK (duration > 0) -- Duration in months
);

CREATE TABLE Students (
    student_id INT PRIMARY KEY,
    student_name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL
);

CREATE TABLE Enrollments (
    enrollment_id INT PRIMARY KEY AUTO_INCREMENT,
    student_id INT,
    course_id INT,
    enrollment_date DATE DEFAULT CURRENT_DATE,
    FOREIGN KEY (student_id) REFERENCES Students(student_id) ON DELETE CASCADE,
    FOREIGN KEY (course_id) REFERENCES Courses(course_id)
);

```

**Tasks:**

1. Insert 3 courses into the Courses table (e.g., "Python Programming", "Data Science", "Web Development").
2. Insert 5 students with valid email values into the Students table.
3. Enroll students into different courses using the Enrollments table.
4. Update a student's email based on their student\_id.
5. Delete a student and check if their enrollments are also removed due to ON DELETE CASCADE.
6. Modify the Courses table to add a NOT NULL constraint on the course\_name column.
7. Use Transactions:
  - a. Begin a transaction.
  - b. Insert a new course.
  - c. Enroll a student in that course.
  - d. ROLLBACK to undo the changes.
  - e. Verify that the rollback worked.

**Mini Project 2: Online Store Order Processing (Using Transactions and Constraints)**

**Project Overview:**

Create an Online Store database where customers can place orders. Ensure data integrity using constraints and handle transactions for order processing.

**Database Schema:**

```
CREATE TABLE Customers (
    customer_id INT PRIMARY KEY,
    customer_name VARCHAR(100) NOT NULL,
```

```
email VARCHAR(100) UNIQUE NOT NULL  
);
```

```
CREATE TABLE Products (  
    product_id INT PRIMARY KEY,  
    product_name VARCHAR(100) UNIQUE NOT NULL,  
    price DECIMAL(10,2) CHECK (price > 0)  
);
```

```
CREATE TABLE Orders (  
    order_id INT PRIMARY KEY,  
    customer_id INT,  
    order_date DATE DEFAULT CURRENT_DATE,  
    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id) ON DELETE  
CASCADE  
);
```

```
CREATE TABLE OrderDetails (  
    order_id INT,  
    product_id INT,  
    quantity INT CHECK (quantity > 0),  
    PRIMARY KEY (order_id, product_id),  
    FOREIGN KEY (order_id) REFERENCES Orders(order_id) ON DELETE CASCADE,  
    FOREIGN KEY (product_id) REFERENCES Products(product_id)  
);
```

#### Tasks:

1. Insert 3 customers and 3 products into the database.
2. Insert an order for a customer and add products to the OrderDetails table.

3. Update a product price and check the impact on existing orders.
4. Delete a customer and ensure their orders are also deleted due to ON DELETE CASCADE.
5. Use Transactions:
  - a. Begin a transaction.
  - b. Insert a new order.
  - c. Add order details.
  - d. SAVEPOINT after inserting products.
  - e. Rollback to SAVEPOINT to remove the last product but keep the order.
  - f. Commit the final changes.

## Day 17

### Indexing and Optimization in SQL

#### Indexes

#### What are Indexes and How Do They Improve Performance?

Indexes are special lookup tables that the database uses to speed up data retrieval. They work like the index of a book, allowing the database to quickly locate rows in a table without scanning every row.

#### Benefits of Indexing:

- Faster SELECT queries
- Improves search operations using WHERE conditions
- Efficient sorting and filtering
- Optimizes JOIN operations

## Downsides of Indexing:

- Increases storage requirements
- Slows down INSERT, UPDATE, DELETE operations due to index maintenance

## Creating and Dropping Indexes

### Syntax to Create an Index:

```
CREATE INDEX index_name ON table_name (column_name);
```

#### Example:

```
CREATE INDEX idx_employee_name ON Employees (last_name);
```

This index improves search speed when querying by last\_name.

#### Dropping an Index:

```
DROP INDEX idx_employee_name ON Employees;
```

## Clustered vs. Non-Clustered Indexes

Feature	Clustered Index	Non-Clustered Index
Storage	Data is stored in the index itself	Stores pointers to actual data
Number per Table	Only 1 per table	Multiple allowed
Performance	Faster for retrieving whole rows	Faster for specific column lookups

### **Example of Clustered Index (Automatically Created on PRIMARY KEY):**

```
CREATE TABLE Employees (
    emp_id INT PRIMARY KEY, -- This is a clustered index
    emp_name VARCHAR(100)
);
```

### **Example of Non-Clustered Index:**

```
CREATE INDEX idx_emp_name ON Employees(emp_name);
```

### **When to Use Indexes for Optimization**

#### **Use Indexes When:**

- Searching frequently on specific columns (WHERE, ORDER BY)
- Performing JOIN operations
- Filtering large datasets

#### **Avoid Indexes When:**

- The table is small (indexing won't help much)
- Columns have high update frequency
- The column has many unique values but is rarely searched

### **Query Optimization**

#### **Analyzing Query Execution Plans (Using EXPLAIN)**

EXPLAIN helps analyze how SQL queries run and how indexes are used.

**Example:**

```
EXPLAIN SELECT * FROM Employees WHERE emp_name = 'John Doe';
```

This returns the execution plan, showing whether indexes are used and how efficiently the query is executed.

**Common Performance Issues and How to Address Them**

Issue	Solution
Full Table Scans	Use indexes
Unoptimized Joins	Ensure JOIN columns are indexed
Excessive Subqueries	Replace with JOINS if possible
Too Many Columns in SELECT	Use SELECT column_name instead of SELECT *

**Using LIMIT to Restrict Results**

To improve performance, use LIMIT to fetch only required records.

**Example:**

```
SELECT * FROM Orders LIMIT 10;
```

This fetches only the first 10 rows, reducing load time.

**Impact of Joins and Subqueries on Performance**

- Joins are faster than subqueries in most cases.
- Use indexes on columns used in joins to improve speed.

### **Example (Optimized JOIN instead of Subquery):**

```
SELECT e.emp_name, d.dept_name
FROM Employees e
JOIN Departments d ON e.dept_id = d.dept_id;
```

## **Normalization and Denormalization**

### **Database Normalization (1NF, 2NF, 3NF)**

Normalization organizes data to **reduce redundancy** and **improve consistency**.

#### **Normalization Levels:**

- **1NF (First Normal Form):** Ensure atomicity (no duplicate columns or repeating groups).
- **2NF (Second Normal Form):** Remove partial dependencies (each column must depend on the whole primary key).
- **3NF (Third Normal Form):** Remove transitive dependencies (non-key columns should not depend on other non-key columns).

### **Example of Normalization (Before and After):**

#### **Before Normalization (Repeating Data):**

OrderID	CustomerName	Product	Quantity
1	Alice	Laptop	2
1	Alice	Mouse	1
2	Bob	Keyboard	1

**After Normalization (Dividing into Two Tables):****Orders Table:**

OrderID	CustomerName
1	Alice
2	Bob

**OrderDetails Table:**

OrderID	Product	Quantity
1	Laptop	2
1	Mouse	1
2	Keyboard	1

**Denormalization and When to Use It**

Denormalization **adds redundancy** to improve performance by reducing JOINs.

**Use Cases for Denormalization:**

- Data is read frequently but updated rarely (e.g., reporting systems).
- Avoiding complex joins in high-performance applications.

**Example of Denormalized Table (Combining Data):**

OrderID	CustomerName	Product	Quantity
1	Alice	Laptop	2
1	Alice	Mouse	1
2	Bob	Keyboard	1

Here, the data is duplicated, but queries will be **faster** as no joins are required.

## Summary of Key Concepts

Concept	Purpose	Example
<b>Indexes</b>	Speed up searches	CREATE INDEX idx_emp ON Employees(emp_name);
<b>EXPLAIN</b>	Analyzes query execution	EXPLAIN SELECT * FROM Orders;
<b>LIMIT</b>	Restrict results	SELECT * FROM Customers LIMIT 5;
<b>Normalization</b>	Reduce redundancy	Splitting tables (1NF, 2NF, 3NF)
<b>Denormalization</b>	Improve performance	Combining tables for faster reads

## Mini Project 1: E-Commerce Database Optimization

### Objective:

Optimize an e-commerce database by implementing indexes, query optimizations, and normalization techniques to improve performance.

### Steps to Complete:

#### 1. Create an E-Commerce Database with Products, Orders, and Customers

```
CREATE TABLE Customers (
    customer_id INT PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100) UNIQUE
);
```

```
CREATE TABLE Products (
```

```
product_id INT PRIMARY KEY,  
name VARCHAR(100),  
price DECIMAL(10,2)  
);  
  
CREATE TABLE Orders (  
    order_id INT PRIMARY KEY,  
    customer_id INT,  
    order_date DATE,  
    total_amount DECIMAL(10,2),  
    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id)  
);
```

## 2. Insert Sample Data

```
INSERT INTO Customers VALUES (1, 'Alice', 'alice@example.com');  
INSERT INTO Customers VALUES (2, 'Bob', 'bob@example.com');
```

```
INSERT INTO Products VALUES (101, 'Laptop', 1200.00);  
INSERT INTO Products VALUES (102, 'Mouse', 25.00);
```

```
INSERT INTO Orders VALUES (201, 1, '2024-03-01', 1225.00);  
INSERT INTO Orders VALUES (202, 2, '2024-03-02', 1200.00);
```

## 3. Create Indexes to Speed Up Queries

```
CREATE INDEX idx_customer_email ON Customers(email);  
CREATE INDEX idx_order_date ON Orders(order_date);
```

#### 4. Use EXPLAIN to Analyze Query Performance

```
EXPLAIN SELECT * FROM Orders WHERE order_date = '2024-03-01';
```

#### 5. Normalize Tables to Reduce Data Redundancy

- Create an **OrderDetails** table to avoid storing product details in the Orders table.

```
CREATE TABLE OrderDetails (
    order_detail_id INT PRIMARY KEY,
    order_id INT,
    product_id INT,
    quantity INT,
    FOREIGN KEY (order_id) REFERENCES Orders(order_id),
    FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

#### 6. Optimize Query Performance Using Joins & LIMIT

```
SELECT c.name, o.order_date, p.name AS product_name
FROM Customers c
JOIN Orders o ON c.customer_id = o.customer_id
JOIN OrderDetails od ON o.order_id = od.order_id
JOIN Products p ON od.product_id = p.product_id
WHERE c.name = 'Alice'
LIMIT 5;
```

## Mini Project 2: Employee Management System Optimization

### Objective:

Improve query performance in an Employee Management System by using indexes, query execution plans, and normalization techniques.

### Steps to Complete:

#### 1. Create an Employee Database with Departments

```
CREATE TABLE Departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(100)
);
```

```
CREATE TABLE Employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100),
    salary DECIMAL(10,2),
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES Departments(dept_id)
);
```

#### 2. Insert Sample Data

```
INSERT INTO Departments VALUES (1, 'HR'), (2, 'Engineering'), (3, 'Sales');
INSERT INTO Employees VALUES (101, 'John Doe', 60000, 2);
INSERT INTO Employees VALUES (102, 'Jane Smith', 75000, 2);
INSERT INTO Employees VALUES (103, 'Mark Lee', 50000, 1);
```

### 3. Create Indexes to Improve Search Performance

```
CREATE INDEX idx_emp_name ON Employees(emp_name);  
CREATE INDEX idx_dept_id ON Employees(dept_id);
```

### 4. Analyze Performance Using EXPLAIN

```
EXPLAIN SELECT * FROM Employees WHERE emp_name = 'John Doe';
```

### 5. Normalize Data to Avoid Redundancy

- Move salary data to a separate table for better salary history tracking.

```
CREATE TABLE Salaries (  
    salary_id INT PRIMARY KEY,  
    emp_id INT,  
    salary DECIMAL(10,2),  
    salary_date DATE,  
    FOREIGN KEY (emp_id) REFERENCES Employees(emp_id)  
);
```

### 6. Optimize Query Performance Using Joins & LIMIT

```
SELECT e.emp_name, d.dept_name, s.salary  
FROM Employees e  
JOIN Departments d ON e.dept_id = d.dept_id  
JOIN Salaries s ON e.emp_id = s.emp_id  
ORDER BY s.salary DESC  
LIMIT 5;
```

## Dataset: Employee and Sales Database

We'll use a dataset for an Employee and Sales Management System to demonstrate indexing and optimization techniques.

Tables in the Dataset

1. **Employees:** Stores employee details
2. **Departments:** Stores department details
3. **Salaries:** Tracks salary history
4. **Sales:** Stores sales records

-- Employee Table

```
CREATE TABLE Employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100),
    job_title VARCHAR(100),
    dept_id INT,
    hire_date DATE
);
```

-- Departments Table

```
CREATE TABLE Departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(100)
);
```

-- Salaries Table

```
CREATE TABLE Salaries (
    salary_id INT PRIMARY KEY,
    emp_id INT,
    salary DECIMAL(10,2),
```

```

salary_date DATE,
FOREIGN KEY (emp_id) REFERENCES Employees(emp_id)
);

```

-- Sales Table

```

CREATE TABLE Sales (
    sale_id INT PRIMARY KEY,
    emp_id INT,
    sale_amount DECIMAL(10,2),
    sale_date DATE,
    FOREIGN KEY (emp_id) REFERENCES Employees(emp_id)
);

```

## **Day 17 Tasks :**

1. Create an index on the emp\_name column in the Employees table. How does it improve search performance?
2. Drop an existing index from the emp\_name column and observe performance changes. What impact does it have on query execution time?
3. Compare clustered and non-clustered indexes by creating a clustered index on emp\_id and a non-clustered index on job\_title. How do they differ in terms of storage and retrieval speed?
4. Create an index on the sale\_date column in the Sales table. How does it improve filtering queries when searching for sales within a specific date range?
5. Use the EXPLAIN statement to analyze the execution plan of a query that retrieves employees by name before and after indexing. What differences do you observe?
6. Optimize a slow query by adding an index on the appropriate column and compare execution times before and after indexing. How much improvement is seen?

7. Use the LIMIT clause to optimize a query that retrieves the top 5 highest salaries. How does LIMIT help in improving query performance?
8. Analyze the performance impact of a JOIN query between Employees and Departments before and after adding indexes. How does indexing affect query execution time?
9. Optimize a query that uses a subquery by adding an index to the filtering column. How does indexing improve subquery performance?
10. Normalize the Employees table by moving job titles to a separate JobTitles table (1NF). Why is this step necessary?
11. Ensure 2NF by creating a separate table for departments to avoid data redundancy. What are the benefits of doing this?
12. Verify if the database follows 3NF by checking if non-key attributes depend only on primary keys. What modifications are needed if a violation is found?
13. Apply denormalization by merging Sales and Employees into a single view for faster data retrieval. What are the advantages and disadvantages of denormalization?

## **Mini Project 1: Library Management System Optimization**

### **Project Objective:**

Optimize the performance of a Library Management System by applying indexing, query optimization, and normalization techniques.

### **Dataset:**

Create a database LibraryDB with the following tables:

1. Books (book\_id, title, author\_id, genre, published\_year)
2. Authors (author\_id, author\_name, country)
3. Borrowers (borrower\_id, borrower\_name, email, membership\_date)

4. BorrowedBooks (borrow\_id, book\_id, borrower\_id, borrow\_date, return\_date)

## Day 17 Tasks:

### 1. Indexing for Faster Searches

- Create a clustered index on book\_id (Primary Key).
- Add a non-clustered index on title to improve search performance.
- Drop the non-clustered index and analyze query performance changes.

### 2. Query Optimization

- Use EXPLAIN to analyze query execution plans before and after indexing.
- Optimize a slow JOIN query between Books and BorrowedBooks by indexing book\_id.

### 3. Using LIMIT for Efficiency

- Retrieve the 5 most recently borrowed books using ORDER BY borrow\_date DESC LIMIT 5.
- Optimize the query by ensuring an index on borrow\_date.

### 4. Normalization for Data Integrity

- Normalize the Books table by creating a separate Genres table (1NF).
- Move author\_name to the Authors table and reference it with a Foreign Key (2NF).

### 5. Denormalization for Fast Reports

- Create a denormalized view combining Books, Authors, and BorrowedBooks to quickly fetch borrower details for overdue books.

## Mini Project 2: E-Commerce Order Management Optimization

### Project Objective:

Improve the efficiency of an E-Commerce Order Management System using indexes, query optimization, and normalization techniques.

### Dataset:

Create a database EcommerceDB with the following tables:

1. Orders (order\_id, customer\_id, order\_date, total\_amount)
2. Customers (customer\_id, customer\_name, email, city)
3. Products (product\_id, product\_name, category\_id, price)
4. OrderDetails (order\_detail\_id, order\_id, product\_id, quantity, subtotal)
5. Categories (category\_id, category\_name)

### Day 17 Tasks:

#### 1. Indexing for Performance Improvement

- Create an index on order\_date to speed up order history retrieval.
- Add a non-clustered index on customer\_name for faster customer searches.

#### 2. Query Optimization

- Use EXPLAIN to analyze query execution for fetching orders with customer details.
- Optimize JOIN queries between Orders, OrderDetails, and Customers by indexing customer\_id and order\_id.

### 3. Using LIMIT for Quick Results

- Retrieve the top 10 highest-value orders using ORDER BY total\_amount DESC LIMIT 10.
- Ensure an index exists on total\_amount for faster execution.

### 4. Normalization for Better Data Structure

- Move category\_name to a separate Categories table (2NF).
- Ensure OrderDetails only references valid order\_id and product\_id using Foreign Keys (3NF).

### 5. Denormalization for Fast Reports

- Create a denormalized table combining Orders, Customers, and OrderDetails to improve dashboard performance for sales analysis.

## Day 18

### Advanced SQL Techniques

#### 1. Views

##### What are Views in SQL?

A view is a virtual table based on the result of a SQL query. It does not store data physically but retrieves data from the underlying tables when queried. Views are used for:

- ✓ Simplifying complex queries
- ✓ Enhancing security by restricting access to specific columns or rows
- ✓ Improving abstraction by providing a different representation of data

### Syntax: Creating a View

```
CREATE VIEW view_name AS  
SELECT column1, column2 FROM table_name  
WHERE condition;
```

**Example:** Creating a view to display high-salary employees:

```
CREATE VIEW HighSalaryEmployees AS  
SELECT emp_id, emp_name, salary  
FROM Employees  
WHERE salary > 50000;
```

### Updating a View

```
CREATE OR REPLACE VIEW view_name AS  
SELECT column1, column2 FROM table_name WHERE condition;
```

**Example:** Updating the view to include department details:

```
CREATE OR REPLACE VIEW HighSalaryEmployees AS  
SELECT emp_id, emp_name, salary, department  
FROM Employees  
WHERE salary > 50000;
```

### Deleting a View

```
DROP VIEW view_name;
```

**Example:**

```
DROP VIEW HighSalaryEmployees;
```

## Using Views for Security & Abstraction

- Restrict sensitive columns (e.g., hide salary details from unauthorized users).
- Provide read-only access to specific data.

**Example:** Creating a view to hide employee salaries from general users:

```
CREATE VIEW EmployeeInfo AS  
SELECT emp_id, emp_name, department FROM Employees;
```

## 2. Stored Procedures and Functions

### Introduction to Stored Procedures & Functions

- Stored Procedures: Predefined SQL code that executes a set of commands.
- Functions: Similar to procedures but must return a value.

### Creating a Stored Procedure

```
CREATE PROCEDURE procedure_name()  
BEGIN  
    -- SQL statements  
END;
```

**Example:** Creating a stored procedure to fetch employee details:

```
CREATE PROCEDURE GetEmployees()  
BEGIN  
    SELECT * FROM Employees;  
END;
```

## Calling a Stored Procedure

```
CALL GetEmployees();
```

## Using Input & Output Parameters

**Example:** Procedure to get employees by department:

```
CREATE PROCEDURE GetEmployeesByDept(IN dept_name VARCHAR(50))
BEGIN
    SELECT * FROM Employees WHERE department = dept_name;
END;
```

Call it using:

```
CALL GetEmployeesByDept('IT');
```

## Difference Between Procedures & Functions

Feature	Stored Procedure	Function
Returns Value?	No (but can return result sets)	Yes (must return a value)
Used In Queries?	No	Yes
Can Modify Data?	Yes	No
Called Using	CALL statement	SELECT statement

**Example of a Function:** Returning total employees in a department:

```
CREATE FUNCTION EmployeeCount(dept_name VARCHAR(50)) RETURNS INT
DETERMINISTIC
BEGIN
    DECLARE total INT;

```

```
SELECT COUNT(*) INTO total FROM Employees WHERE department =  
dept_name;  
RETURN total;  
END;
```

Call it using:

```
SELECT EmployeeCount('HR');
```

### 3. Triggers

What are Triggers and When to Use Them?

Triggers are automated SQL operations that execute when an event (INSERT, UPDATE, DELETE) occurs in a table.

◆ Use cases:

- Enforcing business rules (e.g., auto-update stock after a purchase)
- Maintaining audit logs
- Preventing invalid transactions

#### Creating a Trigger (BEFORE, AFTER)

```
CREATE TRIGGER trigger_name  
BEFORE|AFTER INSERT|UPDATE|DELETE  
ON table_name  
FOR EACH ROW  
BEGIN  
-- SQL statements  
END;
```

**Example:** Automatically logging new employees into an audit table:

```
CREATE TRIGGER logNewEmployee  
AFTER INSERT ON Employees  
FOR EACH ROW  
BEGIN  
    INSERT INTO Employee_Audit (emp_id, action, action_time)  
    VALUES (NEW.emp_id, 'INSERT', NOW());  
END;
```

### **Managing Triggers**

- **Delete a Trigger:**

```
DROP TRIGGER trigger_name;
```

**Example:**

```
DROP TRIGGER logNewEmployee;
```

## **Mini Project 1: Hospital Patient Management System**

### **Project Overview:**

Develop a Hospital Patient Management System where:

- ✓ Views restrict access to confidential patient data.
- ✓ Stored procedures handle patient admissions and discharges.
- ✓ Triggers maintain an audit log for patient updates.

**Tasks:**

- Create a "Patients" table** with columns: patient\_id, name, age, diagnosis, admission\_date, discharge\_date.

```
CREATE TABLE Patients (
    patient_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100),
    age INT,
    diagnosis VARCHAR(255),
    admission_date DATE,
    discharge_date DATE NULL
);
```

- Create a view "PublicPatients"** that hides sensitive diagnosis details.

```
CREATE VIEW PublicPatients AS
SELECT patient_id, name, age, admission_date, discharge_date FROM Patients;
```

- Create a stored procedure "AdmitPatient"** to insert new patients.

```
CREATE PROCEDURE AdmitPatient(
    IN pname VARCHAR(100),
    IN page INT,
    IN pdiagnosis VARCHAR(255),
    IN admission DATE
)
BEGIN
    INSERT INTO Patients(name, age, diagnosis, admission_date)
    VALUES (pname, page, pdiagnosis, admission);
END;
```

**Call Procedure:**

```
CALL AdmitPatient('Alice Smith', 32, 'Pneumonia', '2023-07-10');
```

- 4. Create a trigger "TrackPatientUpdates" to log changes in the Patient\_Audit table.**

```
CREATE TABLE Patient_Audit (
    audit_id INT AUTO_INCREMENT PRIMARY KEY,
    patient_id INT,
    old_diagnosis VARCHAR(255),
    new_diagnosis VARCHAR(255),
    changed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
CREATE TRIGGER TrackPatientUpdates
BEFORE UPDATE ON Patients
FOR EACH ROW
BEGIN
    INSERT INTO Patient_Audit (patient_id, old_diagnosis, new_diagnosis)
    VALUES (OLD.patient_id, OLD.diagnosis, NEW.diagnosis);
END;
```

- 5. Test the trigger by updating a patient's diagnosis.**

```
UPDATE Patients SET diagnosis = 'Asthma' WHERE patient_id = 1;
SELECT * FROM Patient_Audit;
```

## Mini Project 2: Library Management System

### Project Overview:

Build a Library Management System where:

- ✓ Views manage book borrowing details.
- ✓ Stored procedures handle book check-in and check-out.
- ✓ Triggers automatically update book availability.

### Tasks:

#### 1. Create tables "Books" and "Borrowed\_Books".

```
CREATE TABLE Books (
    book_id INT PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(255),
    author VARCHAR(100),
    available_copies INT
);
CREATE TABLE Borrowed_Books (
    borrow_id INT PRIMARY KEY AUTO_INCREMENT,
    book_id INT,
    borrower_name VARCHAR(100),
    borrow_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    return_date DATE NULL,
    FOREIGN KEY (book_id) REFERENCES Books(book_id)
);
```

#### 2. Create a view "AvailableBooks" that shows only books with available copies.

```
CREATE VIEW AvailableBooks AS
SELECT book_id, title, author, available_copies FROM Books WHERE
```

available\_copies > 0;

**3. Create a stored procedure "BorrowBook" to insert book borrow records.**

```
CREATE PROCEDURE BorrowBook(
    IN bookID INT,
    IN borrower VARCHAR(100)
)
BEGIN
    INSERT INTO Borrowed_Books(book_id, borrower_name)
    VALUES (bookID, borrower);
END;
```

**Call Procedure:**

```
CALL BorrowBook(2, 'John Doe');
```

**4. Create a trigger "UpdateBookAvailability" to reduce available copies when a book is borrowed.**

```
CREATE TRIGGER UpdateBookAvailability
AFTER INSERT ON Borrowed_Books
FOR EACH ROW
BEGIN
    UPDATE Books
    SET available_copies = available_copies - 1
    WHERE book_id = NEW.book_id;
END;
```

5. **Test the trigger** by borrowing a book and checking available copies.

```
INSERT INTO Borrowed_Books(book_id, borrower_name) VALUES (1, 'Jane Doe');  
SELECT * FROM Books;
```

## Day 18 Tasks

1. Create a view called EmployeeView that displays only the id, name, and department from an Employees table.
2. Modify the view to include an additional column salary from the Employees table.
3. Delete the view EmployeeView from the database.
4. Create a view HighSalaryEmployees that only shows employees with a salary above 50,000 for security purposes.
5. Create a stored procedure GetEmployeeDetails that retrieves employee details based on employee\_id.
6. Modify the stored procedure to accept an additional parameter for filtering employees by department.
7. Create a stored procedure AddNewEmployee to insert a new record into the Employees table.
8. Create a function CalculateAnnualSalary that takes monthly\_salary as input and returns the annual salary.
9. Differentiate between a stored procedure and a function by implementing both and comparing the results.
10. Create a trigger TrackSalaryChanges that logs salary updates in an Audit\_Salary table before any update occurs.
11. Create a trigger PreventNegativeSalary that prevents inserting or updating an employee's salary to a negative value.

12. Create an AFTER INSERT trigger UpdateDepartmentCount that updates the Department table to reflect the number of employees whenever a new employee is added.
13. Test and analyze all created views, stored procedures, and triggers with sample data, and document their impact on database operations.

## **Mini Project 1: Employee Payroll Management System**

### **Objective:**

Develop an Employee Payroll Management System that uses views for salary details, stored procedures for salary processing, and triggers for automatic tax deductions.

### **Tasks:**

#### 1. Create tables:

- Employees (employee\_id, name, department, salary, hire\_date, tax\_percentage)
- Payroll (payroll\_id, employee\_id, basic\_salary, deductions, net\_salary, payment\_date)

2. Create a view EmployeeSalaryView to display employee\_id, name, department, basic\_salary, deductions, and net\_salary.

3. Write a stored procedure ProcessSalary(employee\_id) to calculate net salary after tax deductions and insert data into the Payroll table.

4. Write a stored procedure UpdateSalary(employee\_id, new\_salary) to modify an employee's salary and reflect the changes in payroll processing.

5. Implement a trigger AutoDeductTax to automatically calculate and update deductions based on tax percentage before salary is processed.

6. Implement a trigger PreventNegativeSalary to prevent net salary from being negative due to excessive deductions.
7. Test the views, stored procedures, and triggers with sample employee salary data.

## **Mini Project 2: Online Order Management System**

### **Objective:**

Build an Online Order Management System that uses views for order details, stored procedures for order processing, and triggers for inventory updates.

### **Tasks:**

#### 1. Create tables:

- Customers (customer\_id, name, email, phone)
- Products (product\_id, name, price, stock\_quantity)
- Orders (order\_id, customer\_id, order\_date, total\_amount, status)
- OrderDetails (order\_detail\_id, order\_id, product\_id, quantity, subtotal)

2. Create a view CustomerOrdersView to display customer\_id, name, order\_id, order\_date, total\_amount, and status.

3. Write a stored procedure PlaceOrder(customer\_id, product\_id, quantity) to insert an order and calculate the total amount.

4. Write a stored procedure CancelOrder(order\_id) to update the order status to 'Cancelled' and restock the items.

5. Implement a trigger UpdateStockAfterOrder to automatically reduce product stock when an order is placed.

6. Implement a trigger PreventOrderIfOutOfStock to prevent orders from being placed if the requested product is out of stock.
7. Test the views, stored procedures, and triggers with sample customer and order data.

## Day 19

### SQL for Reporting and Advanced Data Analysis

In this section, we will cover advanced SQL techniques used for reporting and data analysis, including hierarchical data management, window functions, and common table expressions (CTEs).

#### 1. Working with Hierarchical Data

##### **Definition:**

Hierarchical data represents relationships where entities are arranged in a tree-like structure. Examples include:

- Organizational charts
- Category and subcategory structures
- File system directories

SQL provides recursive queries to traverse hierarchical data using:

- CONNECT BY (Oracle)
- WITH RECURSIVE (PostgreSQL, MySQL, SQL Server)

## **Example 1: Managing Employee Hierarchy**

### **Schema:**

```
CREATE TABLE Employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100),
    manager_id INT REFERENCES Employees(emp_id) -- Self-referencing foreign key
);
```

### **Inserting Data:**

```
INSERT INTO Employees (emp_id, emp_name, manager_id) VALUES (1, 'CEO', NULL);
INSERT INTO Employees (emp_id, emp_name, manager_id) VALUES (2, 'Manager A', 1);
INSERT INTO Employees (emp_id, emp_name, manager_id) VALUES (3, 'Manager B', 1);
INSERT INTO Employees (emp_id, emp_name, manager_id) VALUES (4, 'Employee X', 2);
INSERT INTO Employees (emp_id, emp_name, manager_id) VALUES (5, 'Employee Y', 2);
```

### **Using WITH RECURSIVE for Hierarchical Queries (PostgreSQL, MySQL, SQL Server)**

```
WITH RECURSIVE EmployeeHierarchy AS (
    SELECT emp_id, emp_name, manager_id, 1 AS level
    FROM Employees
    WHERE manager_id IS NULL -- Start from the top level (CEO)

    UNION ALL
```

```

SELECT e.emp_id, e.emp_name, e.manager_id, eh.level + 1
FROM Employees e
JOIN EmployeeHierarchy eh ON e.manager_id = eh.emp_id
)
SELECT * FROM EmployeeHierarchy;

```

**Output:**

emp_id	emp_name	manager_id	level
1	CEO	NULL	1
2	Manager A	1	2
3	Manager B	1	2
4	Employee X	2	3
5	Employee Y	2	3

This retrieves all employees in a hierarchical manner.

## 2. Window Functions

**Definition:**

Window functions perform calculations across a set of rows related to the current row without collapsing them into a single result.

They are useful for:

- Ranking rows
- Running totals
- Comparing values from previous rows

## Example 2: Employee Salary Ranking

Schema:

```
CREATE TABLE Employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100),
    department VARCHAR(50),
    salary INT
);
```

```
INSERT INTO Employees VALUES
(1, 'Alice', 'HR', 50000),
(2, 'Bob', 'HR', 60000),
(3, 'Charlie', 'IT', 70000),
(4, 'David', 'IT', 80000),
(5, 'Eve', 'Finance', 55000);
```

Using RANK(), DENSE\_RANK(), and ROW\_NUMBER()

```
SELECT emp_name, department, salary,
    ROW_NUMBER() OVER (ORDER BY salary DESC) AS row_number,
    RANK() OVER (ORDER BY salary DESC) AS rank,
    DENSE_RANK() OVER (ORDER BY salary DESC) AS dense_rank
FROM Employees;
```

**Output:**

emp_name	department	salary	row_number	rank	dense_rank
David	IT	80000	1	1	1

---

David	IT	80000	1	1	1
-------	----	-------	---	---	---

Charlie	IT	70000	2	2	2
Bob	HR	60000	3	3	3
Eve	Finance	55000	4	4	4
Alice	HR	50000	5	5	5

**Difference Between Ranking Functions:**

- ROW\_NUMBER() assigns unique numbers, even for same salary.
- RANK() assigns the same rank to ties but skips the next number.
- DENSE\_RANK() assigns the same rank to ties but does NOT skip numbers.

**Example 3: LEAD() and LAG() Functions**

```
SELECT emp_name, salary,
LAG(salary) OVER (ORDER BY salary DESC) AS previous_salary,
LEAD(salary) OVER (ORDER BY salary DESC) AS next_salary
FROM Employees;
```

**Output:**

emp_name	salary	previous_salary	next_salary
David	80000	NULL	70000
Charlie	70000	80000	60000
Bob	60000	70000	55000
Eve	55000	60000	50000
Alice	50000	55000	NULL

David	80000	NULL	70000
Charlie	70000	80000	60000
Bob	60000	70000	55000
Eve	55000	60000	50000
Alice	50000	55000	NULL

LAG() looks at the previous row, while LEAD() looks at the next row.

### 3. Common Table Expressions (CTEs)

#### Definition:

A CTE (Common Table Expression) is a temporary result set that improves query readability and modularity.

#### Example 4: Using CTE for Readable Queries

```
WITH HighSalaryEmployees AS (
    SELECT emp_name, department, salary
    FROM Employees
    WHERE salary > 55000
)
SELECT * FROM HighSalaryEmployees;
```

This creates a temporary table of employees earning more than 55,000.

#### Example 5: Recursive CTE for Hierarchical Queries

```
WITH RECURSIVE EmployeeHierarchy AS (
    SELECT emp_id, emp_name, manager_id, 1 AS level
    FROM Employees
    WHERE manager_id IS NULL
    UNION ALL
    SELECT e.emp_id, e.emp_name, e.manager_id, eh.level + 1
    FROM Employees e
    JOIN EmployeeHierarchy eh ON e.manager_id = eh.emp_id
)
SELECT * FROM EmployeeHierarchy;
```

Recursive CTEs allow querying tree-like structures efficiently.

## Summary

Feature	Definition	Syntax	Use Case
Hierarchical Queries	Retrieves tree-structured data	WITH RECURSIVE	Employee hierarchy, categories
Window Functions	Perform calculations without collapsing rows	ROW_NUMBER(), RANK(), LEAD(), LAG()	Ranking, trends, comparisons
CTEs	Temporary named result sets	WITH temp_table AS (...)	Readable, modular queries

## Day 19 tasks

Task 1: Create an Employee Hierarchy Table

✓ Create an Employees table with the following columns:

- emp\_id (Primary Key)
- emp\_name
- position
- salary
- manager\_id (Foreign Key referencing emp\_id)

Write an INSERT query to add at least 10 employees with different positions and managers.

Task 2: Retrieve a Full Employee Hierarchy (Recursive Query)

✓ Use WITH RECURSIVE (PostgreSQL, MySQL 8+) or CONNECT BY (Oracle) to display the employee hierarchy, showing:

- Employee Name
- Position

- Manager Name
- Hierarchy Level

Order the result to display the hierarchy from top (CEO) to bottom (Interns).

#### Task 3: Find Employees Reporting to a Specific Manager

✓ Modify the recursive query to show only employees under a specific manager (e.g., manager\_id = 2).

Display employees along with their hierarchy level and manager's name.

#### Task 4: Rank Employees by Salary Using RANK()

✓ Use a window function (RANK()) to assign ranks to employees based on salary in descending order.

If two employees have the same salary, they should have the same rank.

#### Task 5: Rank Employees by Salary Using DENSE\_RANK()

✓ Modify the previous query to use DENSE\_RANK() instead of RANK().

Compare the difference between RANK() and DENSE\_RANK().

#### Task 6: Assign Row Numbers to Employees Using ROW\_NUMBER()

✓ Use ROW\_NUMBER() to give a unique row number to each employee ordered by salary.

Explain the difference between ROW\_NUMBER(), RANK(), and DENSE\_RANK().

#### Task 7: Categorize Employees into Salary Groups Using NTILE()

Use NTILE(4) to divide employees into 4 salary groups (Quartiles).

Show the salary range for each group and which employees belong to which quartile.

Task 8: Find Previous and Next Salaries Using LAG() and LEAD()

✓ Use LAG() and LEAD() to display:

- The previous employee's salary (using LAG())
- The next employee's salary (using LEAD())

Order the results by salary and show comparisons.

Task 9: Compare Employee Salaries Over Time (Month-to-Month Analysis)

✓ Given a Salaries table with columns:

- emp\_id
- salary\_amount
- salary\_date

Use LAG() to compare each employee's current salary with the previous month's salary.

Task 10: Create a CTE to Filter Employees by Salary

✓ Write a Common Table Expression (CTE) to:

- Filter employees with a salary greater than 80,000
- Show their names, positions, and salaries

Use the CTE in a SELECT query to retrieve results.

Task 11: Create a Recursive CTE for an Organization Hierarchy

✓ Use a recursive CTE to:

- Display all employees along with their reporting hierarchy
- Show who reports to whom and their level in the hierarchy

Compare the recursive CTE with a normal SQL join query.

#### Task 12: Find Top 3 Highest-Paid Employees in Each Department

✓ Given an Employees table with a department column, use:

- PARTITION BY department
- ORDER BY salary DESC
- RANK() or DENSE\_RANK()

Retrieve only the top 3 highest-paid employees per department.

#### Task 13: Generate a Running Total of Employee Salaries

✓ Use a window function with SUM() to generate a running total of employee salaries ordered by salary amount.

Explain how this is different from a GROUP BY aggregate function.

### **Summary of SQL Techniques Used in These Tasks:**

SQL Concept	Tasks Covered
Recursive Queries	Task 2, Task 3, Task 11
Window Functions	Task 4 - Task 9, Task 12, Task 13
CTEs (Common Table Expressions)	Task 10, Task 11
Ranking Functions (RANK(), DENSE_RANK(), ROW_NUMBER())	Task 4 - Task 6, Task 12
NTILE() for Quartiles	Task 7
LEAD() and LAG() for Trends	Task 8, Task 9
PARTITION BY for Grouped Rankings	Task 12

## Mini Project 1: University Course Enrollment Analysis

### Objective:

Develop an SQL-based reporting system to analyze student enrollments, course hierarchy, and academic performance using hierarchical queries, window functions, and CTEs.

### Project Steps:

1. Create the following tables:

- Students (student\_id, student\_name, batch, department)
- Courses (course\_id, course\_name, prerequisite\_course\_id) (Self-referencing to form a hierarchy)
- Enrollments (enrollment\_id, student\_id, course\_id, grade)

2. Insert sample data:

- At least 10 courses with prerequisites (e.g., "Database Systems" → "Advanced SQL").
- At least 30 students enrolled in multiple courses with grades assigned.

3. Implement a recursive query to display course dependencies using:

- WITH RECURSIVE (PostgreSQL, MySQL 8+)
- CONNECT BY (Oracle)

4. Use window functions to:

- Rank students based on their performance in each course (RANK(), DENSE\_RANK()).
- Find the previous and next highest grades per course (LAG(), LEAD()).

5. Use a CTE to:

- Identify students who have completed all prerequisite courses for an advanced course.
- Find the average grade per department.

**Final Output:**

- Course dependency hierarchy.
- Student ranking and performance analysis.
- Eligibility check for advanced courses.

## **Mini Project 2: E-Commerce Sales Performance Tracker**

**Objective:**

Build an SQL-based sales analytics dashboard for an e-commerce store using window functions, CTEs, and hierarchical queries.

**Project Steps:**

1. Create the following tables:

- Products (product\_id, product\_name, category\_id, price)
- Categories (category\_id, category\_name, parent\_category\_id) (Self-referencing to form a hierarchy)
- Orders (order\_id, customer\_id, order\_date, total\_amount)
- Order\_Items (order\_item\_id, order\_id, product\_id, quantity, subtotal)

2. Insert sample data:

- At least 10 product categories with parent-child relationships (e.g., "Electronics" → "Laptops").
- 50+ orders from various customers over multiple months.

3. Implement a recursive query to display the category hierarchy using:

- WITH RECURSIVE (PostgreSQL, MySQL 8+)
- CONNECT BY (Oracle)

4. Use window functions to:

- Rank products by total sales in each category (RANK(), DENSE\_RANK()).
- Compare sales trends of the same product over time (LAG(), LEAD()).
- Divide products into 4 sales performance tiers (NTILE(4)).

5. Use a CTE to:

- Identify top-selling categories based on sales volume.
- Find the most frequently purchased product in each category.

**Final Output:**

- Product category hierarchy visualization.
- Sales ranking and performance analysis.
- Category-wise top-selling products.

**Key Skills Practiced:**

- ✓ Hierarchical Queries: Recursive CTEs, WITH RECURSIVE, CONNECT BY.
- ✓ Window Functions: RANK(), DENSE\_RANK(), LEAD(), LAG(), NTILE().
- ✓ CTEs: Readable and modular SQL queries for ranking and filtering.
- ✓ Data Analysis & Reporting: Academic performance tracking, e-commerce sales analytics.

# Day 20

## SQL for Data Warehousing and Business Intelligence

Data Warehousing and Business Intelligence (BI) involve managing and analyzing large datasets for decision-making. SQL plays a crucial role in data storage, transformation, aggregation, and reporting within a data warehouse environment.

### 1. Data Warehousing Concepts

#### What is Data Warehousing?

A Data Warehouse (DW) is a central repository where data from multiple sources is collected, stored, and analyzed for business intelligence and reporting. It is optimized for read-heavy operations and structured for efficient querying.

#### Key Characteristics:

- Stores historical data for analysis.
- Supports complex queries and reporting.
- Optimized for fast read operations (unlike transactional databases).

#### OLAP vs. OLTP

Feature	OLTP (Online Transaction Processing)	OLAP (Online Analytical Processing)
Purpose	Fast transactions (Insert, Update, Delete)	Complex analytical queries
Data Type	Real-time, detailed	Historical, aggregated
Normalization	Highly normalized (to avoid redundancy)	Denormalized (for faster queries)
Example	Banking transactions, e-commerce orders	Sales analysis, trend forecasting

**Example:**

**OLTP (Transactional Query - Insert Order Data)**

```
INSERT INTO orders (order_id, customer_id, order_date, total_amount)
VALUES (1001, 5, '2025-03-06', 250.00);
```

**OLAP (Analytical Query - Monthly Sales Report)**

```
SELECT EXTRACT(MONTH FROM order_date) AS month, SUM(total_amount) AS
total_sales
FROM orders
GROUP BY month;
```

**Star and Snowflake Schemas**

Schemas define how tables are structured in a data warehouse.

**1. Star Schema:**

- Single Fact Table (e.g., Sales)
- Multiple Dimension Tables (e.g., Products, Customers, Time)
- Faster queries but redundant data.

**Example:**

- Fact\_Sales (sales\_id, product\_id, customer\_id, time\_id, revenue)
- Dim\_Product (product\_id, product\_name, category)
- Dim\_Customer (customer\_id, name, location)
- Dim\_Time (time\_id, month, year)

**2. Snowflake Schema:**

- Dimension tables are normalized into sub-tables.
- Less redundancy but slower queries due to joins.

**Example:**

- Dim\_Product → (product\_id, category\_id)
- Category\_Details (category\_id, category\_name)

### **Example Query (Star Schema - Total Sales Per Category)**

```
SELECT c.category_name, SUM(s.revenue) AS total_sales  
FROM Fact_Sales s  
JOIN Dim_Product p ON s.product_id = p.product_id  
JOIN Category_Details c ON p.category_id = c.category_id  
GROUP BY c.category_name;
```

## **2. Data Aggregation and Reporting**

SQL enables powerful data aggregation techniques for reporting.

### **GROUP BY and HAVING**

Used to group and filter aggregated data.

#### **Example: Total Revenue by Product Category**

```
SELECT p.category, SUM(o.total_amount) AS total_revenue  
FROM orders o  
JOIN products p ON o.product_id = p.product_id  
GROUP BY p.category  
HAVING SUM(o.total_amount) > 5000; -- Filters categories with revenue > 5000
```

### **Generating Summary Reports**

#### **Example: Monthly Sales Performance Report**

```
SELECT EXTRACT(YEAR FROM order_date) AS year,  
       EXTRACT(MONTH FROM order_date) AS month,  
       COUNT(order_id) AS total_orders,  
       SUM(total_amount) AS total_revenue  
FROM orders  
GROUP BY year, month  
ORDER BY year DESC, month DESC;
```

**Business Use Case:** Helps businesses track seasonal trends in sales.

### **3. ETL Process (Extract, Transform, Load)**

ETL (Extract, Transform, Load) is the process of collecting, cleaning, and loading data into a data warehouse.

#### **Extracting Data**

Fetching data from different sources (databases, CSV, APIs).

##### **Example: Extract Active Customers**

```
SELECT customer_id, customer_name, email
FROM customers
WHERE status = 'Active';
```

#### **Transforming Data**

Cleaning and formatting the extracted data before storage.

##### **Example: Standardizing Customer Names**

```
UPDATE customers
SET customer_name = UPPER(customer_name)
WHERE customer_name IS NOT NULL;
```

#### **Loading Data**

Storing transformed data into a Data Warehouse Table.

##### **Example: Inserting Processed Data into Warehouse**

```
INSERT INTO dw_sales (customer_id, total_purchases, last_purchase_date)
SELECT customer_id, SUM(total_amount), MAX(order_date)
FROM orders
GROUP BY customer_id;
```

**Business Use Case:** Helps in customer segmentation for marketing campaigns.

## Summary

Data Warehousing enables storing and analyzing large datasets.

OLTP vs. OLAP – Transactional vs. Analytical processing.

Star vs. Snowflake Schema – Trade-off between speed and normalization.

SQL Reporting – GROUP BY, HAVING, and summary reports for BI.

ETL – Extracting, transforming, and loading data into a warehouse.

## Mini Project 1: Sales Data Warehouse & Reporting System

### Objective:

Build a Sales Data Warehouse with a Star Schema and use SQL to generate business reports.

### Steps to Implement:

#### 1. Create the Data Warehouse Schema (Star Schema)

- Fact\_Sales (sales\_id, product\_id, customer\_id, time\_id, revenue, quantity\_sold)
- Dim\_Product (product\_id, product\_name, category)
- Dim\_Customer (customer\_id, name, location, age\_group)
- Dim\_Time (time\_id, date, month, year, quarter)

#### 2. Insert Sample Data

- Populate Fact\_Sales with sales transactions.
- Populate Dim\_Product, Dim\_Customer, and Dim\_Time with relevant information.

#### 3. Generate Business Reports

- Total Sales per Product Category

```
SELECT p.category, SUM(s.revenue) AS total_sales
FROM Fact_Sales s
JOIN Dim_Product p ON s.product_id = p.product_id
GROUP BY p.category;
```

- Top 5 Customers by Revenue

```
SELECT c.name, SUM(s.revenue) AS total_spent
FROM Fact_Sales s
JOIN Dim_Customer c ON s.customer_id = c.customer_id
GROUP BY c.name
ORDER BY total_spent DESC
LIMIT 5;
```

- Quarterly Revenue Analysis

```
SELECT t.quarter, SUM(s.revenue) AS quarterly_revenue
FROM Fact_Sales s
JOIN Dim_Time t ON s.time_id = t.time_id
GROUP BY t.quarter
ORDER BY t.quarter;
```

#### **Outcome:**

- A structured sales data warehouse with Star Schema.
- Advanced reporting queries for business insights.

## **Mini Project 2: ETL Pipeline for Customer Insights**

#### **Objective:**

Develop an ETL (Extract, Transform, Load) pipeline to process customer data for business intelligence.

### Steps to Implement:

#### 1. Extract Data from a Source Database (Customers & Orders Tables)

```
SELECT customer_id, customer_name, email, join_date, last_order_date,
total_spent
FROM customers
WHERE status = 'Active';
```

#### 2. Transform Data (Clean & Standardize)

- Convert customer names to uppercase for consistency.

```
UPDATE customers
SET customer_name = UPPER(customer_name);
```

- Classify customers based on total spending.

```
SELECT customer_id,
CASE
    WHEN total_spent > 10000 THEN 'Premium'
    WHEN total_spent BETWEEN 5000 AND 9999 THEN 'Gold'
    ELSE 'Regular'
END AS customer_category
FROM customers;
```

#### 3. Load Transformed Data into the Data Warehouse

```
INSERT INTO dw_customers (customer_id, customer_name, email, join_date,
last_order_date, customer_category)
SELECT customer_id, customer_name, email, join_date, last_order_date,
CASE
    WHEN total_spent > 10000 THEN 'Premium'
    WHEN total_spent BETWEEN 5000 AND 9999 THEN 'Gold'
    ELSE 'Regular'
END
```

FROM customers;

### **Outcome:**

- Automated ETL pipeline to process customer data.
- Business Intelligence Reports based on customer segmentation.

### **Dataset: Sales Data Warehouse for Business Intelligence**

The dataset represents a Sales Data Warehouse structured using the Star Schema.

It includes:

- Fact Table: fact\_sales – Sales transactions.
- Dimension Tables:
  - dim\_product – Product details.
  - dim\_customer – Customer details.
  - dim\_time – Time details.
  - dim\_store – Store details.

### **Schema Design (Star Schema)**

#### 1. Fact Table: fact\_sales (Stores sales transactions)

Column Name	Data Type	Description
sales_id	INT (PK)	Unique ID for each sale
product_id	INT (FK)	Product sold
customer_id	INT (FK)	Customer who made the purchase
time_id	INT (FK)	Date of purchase
store_id	INT (FK)	Store where the sale happened
revenue	DECIMAL(10,2)	Total revenue from the sale
quantity_sold	INT	Number of units sold

**2. Dimension Table: dim\_product (Product details)**

Column Name	Data Type	Description
product_id	INT (PK)	Unique ID for product
product_name	VARCHAR(100)	Name of product
category	VARCHAR(50)	Product category
price	DECIMAL(10,2)	Price per unit

**3. Dimension Table: dim\_customer (Customer details)**

Column Name	Data Type	Description
customer_id	INT (PK)	Unique ID for customer
customer_name	VARCHAR(100)	Name of customer
location	VARCHAR(100)	Customer location
age_group	VARCHAR(20)	Age group (e.g., 18-25, 26-40, etc.)

**4. Dimension Table: dim\_time (Time details)**

Column Name	Data Type	Description
time_id	INT (PK)	Unique ID for each date
date	DATE	Calendar date
month	VARCHAR(20)	Month name
year	INT	Year of transaction
quarter	VARCHAR(10)	Quarter (Q1, Q2, Q3, Q4)

**5. Dimension Table: dim\_store (Store details)**

Column Name	Data Type	Description
store_id	INT (PK)	Unique ID for store
store_name	VARCHAR(100)	Name of store
region	VARCHAR(50)	Store region

## Day 20 Tasks

1. Create the Data Warehouse Schema – Write SQL statements to create fact\_sales, dim\_product, dim\_customer, dim\_time, and dim\_store tables using the Star Schema.
2. Insert Sample Data into Dimension Tables – Populate dim\_product, dim\_customer, dim\_time, and dim\_store with at least 10 records each.
3. Insert Sales Transactions into the Fact Table – Populate fact\_sales with at least 20 records, ensuring relationships with dimension tables.
4. Perform OLAP Analysis – Write an SQL query to aggregate total sales revenue by product category using GROUP BY.
5. Identify the Top 5 Best-Selling Products – Retrieve the top 5 products based on total quantity sold.
6. Calculate Monthly Sales Revenue Using Window Functions – Use window functions (OVER, PARTITION BY) to calculate total monthly sales.
7. Find Repeat Customers – Identify customers who have purchased more than once, using HAVING COUNT() > 1.
8. Implement a Recursive Query for Hierarchical Data – Write a recursive Common Table Expression (CTE) to analyze store regions.
9. Identify High-Value Customers – Find customers who have spent more than \$5000 in total purchases.
10. Create an ETL Process: Extract Data from the Sales Table – Write an SQL query to extract sales data for a specific year.
11. Transform Data: Standardize Customer Names – Write an SQL query to convert all customer names to uppercase.
12. Load Transformed Data into a New Table – Insert transformed sales data into a new summary table for reporting.

13. Generate a Sales Summary Report for Business Intelligence – Write an SQL query to generate a yearly sales summary grouped by product category.

## **Mini Project Tasks for SQL in Data Warehousing & Business Intelligence**

### **1. E-Commerce Customer Purchase Behavior Analysis**

Objective: Build a data warehouse for an e-commerce platform to analyze customer purchase behavior and generate insights.

Steps:

- Design a Star Schema with:
  - Fact table: fact\_orders (containing order\_id, customer\_id, product\_id, order\_date, quantity, total\_price).
  - Dimension tables:
    - dim\_customers (customer details, location, registration date).
    - dim\_products (product name, category, price).
    - dim\_time (year, month, week, day for time-based analysis).
- Populate tables with sample data.
- Write analytical queries for:
  - Finding the top 5 products by sales revenue.
  - Identifying high-value customers (based on total purchases).
  - Analyzing monthly and yearly sales trends using window functions.
  - Segmenting customers into different purchase groups using CASE statements.

## 2. ETL Pipeline for Product Inventory Management

Objective: Create an ETL pipeline using SQL for automating inventory updates and generating stock reports.

Steps:

- Extract:
  - Retrieve raw inventory data from raw\_inventory\_logs (containing product\_id, stock\_added, stock\_sold, date).
- Transform:
  - Calculate real-time stock levels by aggregating stock movements.
  - Identify low-stock products (current\_stock < reorder\_threshold).
  - Categorize products as Fast-Moving, Slow-Moving, or Dead Stock using a CASE statement based on sales frequency.
- Load:
  - Insert the cleaned and updated inventory data into a final\_inventory\_status table.
- Generate Reports:
  - Create a daily inventory summary report.
  - Find out-of-stock products that need urgent restocking.
  - Analyze sales trends per product category to optimize inventory purchasing decisions.

# Day 21

## Final Project and Practical Applications: SQL for E-Commerce Database

This project will involve designing a real-world e-commerce database using SQL, inserting at least 20 sample records, and writing advanced queries for reporting and business intelligence (BI).

### Step 1: Understanding the Project Scope

#### Objective:

- Develop an e-commerce database for managing customers, products, orders, payments, and shipments.
- Implement OLAP (Online Analytical Processing) and BI techniques for insightful reports.
- Optimize SQL queries for performance and security.

#### Key Features:

- ✓ Customer Management (Registering users and tracking purchases).
- ✓ Product Catalog (Managing products, categories, and inventory).
- ✓ Order Processing (Handling orders, payments, and shipments).
- ✓ Business Intelligence Reports (Sales trends, customer behavior, and top-selling products).

### Step 2: Database Schema Design

We will follow a **Star Schema** approach with **six tables**:

Table Name	Description
customers	Stores customer details
products	Stores product details

orders	Stores order details
order_details	Links orders with products
payments	Stores payment transactions
shipments	Stores shipping details

## SQL Code: Creating Tables

```
CREATE DATABASE ecommerce;
```

```
USE ecommerce;
```

```
CREATE TABLE customers (
    customer_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100),
    email VARCHAR(100) UNIQUE,
    phone_number VARCHAR(15),
    address TEXT,
    registration_date DATE
);
```

```
CREATE TABLE products (
    product_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100),
    category VARCHAR(50),
    price DECIMAL(10,2),
    stock_quantity INT
);
```

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY AUTO_INCREMENT,
    customer_id INT,
    order_date DATE,
    total_amount DECIMAL(10,2),
    order_status VARCHAR(20),
```

```
FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);
```

```
CREATE TABLE order_details (
    order_detail_id INT PRIMARY KEY AUTO_INCREMENT,
    order_id INT,
    product_id INT,
    quantity INT,
    subtotal_price DECIMAL(10,2),
    FOREIGN KEY (order_id) REFERENCES orders(order_id),
    FOREIGN KEY (product_id) REFERENCES products(product_id)
);
```

```
CREATE TABLE payments (
    payment_id INT PRIMARY KEY AUTO_INCREMENT,
    order_id INT,
    payment_method VARCHAR(50),
    payment_status VARCHAR(20),
    payment_date DATE,
    FOREIGN KEY (order_id) REFERENCES orders(order_id)
);
```

```
CREATE TABLE shipments (
    shipment_id INT PRIMARY KEY AUTO_INCREMENT,
    order_id INT,
    shipment_date DATE,
    delivery_status VARCHAR(50),
    FOREIGN KEY (order_id) REFERENCES orders(order_id)
);
```

## Step 3: Inserting Sample Data (Minimum 20 Records)

### Insert 5 Customers

```
INSERT INTO customers (name, email, phone_number, address, registration_date)
VALUES
('John Doe', 'john@example.com', '9876543210', '123 Main St, NY', '2023-01-10'),
('Jane Smith', 'jane@example.com', '9876504321', '456 Maple St, LA', '2023-02-15'),
('Michael Brown', 'michael@example.com', '9876512345', '789 Oak St, TX', '2023-03-20'),
('Emily Davis', 'emily@example.com', '9876523456', '101 Pine St, FL', '2023-04-05'),
('David Johnson', 'david@example.com', '9876534567', '202 Elm St, IL', '2023-05-10');
```

### Insert 5 Products

```
INSERT INTO products (name, category, price, stock_quantity) VALUES
('Laptop', 'Electronics', 1200.00, 50),
('Smartphone', 'Electronics', 800.00, 100),
('Headphones', 'Accessories', 150.00, 200),
('Tablet', 'Electronics', 600.00, 75),
('Keyboard', 'Accessories', 50.00, 300);
```

### Insert 5 Orders

```
INSERT INTO orders (customer_id, order_date, total_amount, order_status)
VALUES
(1, '2023-06-01', 2000.00, 'Confirmed'),
(2, '2023-06-03', 800.00, 'Shipped'),
(3, '2023-06-05', 950.00, 'Delivered'),
(4, '2023-06-07', 1200.00, 'Confirmed'),
(5, '2023-06-10', 150.00, 'Delivered');
```

## Insert 5 Order Details

```
INSERT INTO order_details (order_id, product_id, quantity, subtotal_price)
VALUES
(1, 1, 1, 1200.00),
(1, 2, 1, 800.00),
(2, 2, 1, 800.00),
(3, 3, 2, 300.00),
(4, 1, 1, 1200.00);
```

## Step 4: Writing Advanced SQL Queries

### 1. Retrieve Customer Purchase History

```
SELECT c.name, o.order_id, o.order_date, o.total_amount, o.order_status
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
ORDER BY o.order_date DESC;
```

### 2. Identify Best-Selling Products

```
SELECT p.name, SUM(od.quantity) AS total_sold
FROM order_details od
JOIN products p ON od.product_id = p.product_id
GROUP BY p.name
ORDER BY total_sold DESC;
```

### 3. Generate Monthly Sales Report Using Window Functions

```
SELECT
EXTRACT(YEAR FROM order_date) AS year,
EXTRACT(MONTH FROM order_date) AS month,
SUM(total_amount) AS monthly_sales,
RANK() OVER (ORDER BY SUM(total_amount) DESC) AS sales_rank
```

```
FROM orders  
GROUP BY year, month  
ORDER BY year DESC, month DESC;
```

#### **4. Customers with the Most Orders**

```
SELECT c.name, COUNT(o.order_id) AS total_orders  
FROM customers c  
JOIN orders o ON c.customer_id = o.customer_id  
GROUP BY c.name  
ORDER BY total_orders DESC;
```

### **Step 5: Final Review & Best Practices**

#### **Performance Optimization:**

- Use Indexes for frequently queried columns (customer\_id, product\_id).
- Optimize JOIN operations to retrieve only necessary data.
- Use LIMIT to restrict large dataset queries.

#### **Security Against SQL Injection:**

- Use prepared statements when taking user input.
- Avoid dynamic SQL.
- Restrict database access for non-admin users.

#### **Business Intelligence Insights:**

- Top-selling products
- High-value customers
- Monthly revenue trends
- Order fulfillment status

## Final Outcome: Real-World Application

After completing this project, you will have:

- ✓ A fully functional e-commerce database with optimized queries.
- ✓ Advanced BI reports on sales, customer behavior, and product trends.
- ✓ SQL security best practices implemented.

# Basics of Python

## Day 22

### Python Setup, Introduction, and print() Function

#### Introduction to Python

Python is a high-level, interpreted programming language that is known for its simplicity, readability, and versatility. It is widely used for:

- ✓ Web development (Django, Flask)
- ✓ Data Science & Machine Learning (Pandas, NumPy, TensorFlow)
- ✓ Automation & Scripting
- ✓ Game Development (Pygame)
- ✓ Cybersecurity & Ethical Hacking

Python follows a clear and readable syntax that makes it an excellent choice for both beginners and advanced developers.

## Setting Up Python on Your System

Before you start coding in Python, you need to install it on your computer.

### Step 1: Check if Python is Installed

Open the terminal (Command Prompt or PowerShell on Windows, Terminal on Mac/Linux) and type:

```
python --version
```

If Python is installed, you will see something like:

```
Python 3.10.6
```

If not, proceed to install Python.

### Step 2: Download and Install Python

1. Go to the official Python website: <https://www.python.org/downloads/>
2. Download the latest Python 3.x.x version.
3. Run the installer and check the box "Add Python to PATH" before clicking Install.

### Step 3: Verify Installation

After installation, open a terminal and type:

```
python --version
```

or

```
python3 --version
```

If you see the version number, Python is successfully installed.

## Running Python Code

There are three ways to run Python code:

### 1. Using the Python Interactive Shell

- Open a terminal and type python or python3
- You'll see >>>, meaning you can type Python commands directly.
- Example: >>> print("Hello, Python!")  
Hello, Python!
- To exit, type exit() or press Ctrl + Z (Windows) or Ctrl + D (Mac/Linux).

### 2. Using a Python Script (.py file)

- Open a text editor (VS Code, PyCharm, or Notepad).
- Write Python code in a file, e.g., hello.py: print("Hello, World!")
- Save it as hello.py and run it in the terminal: python hello.py
- Output: Hello, World!

### 3. Using an Online Compiler

- You can run Python code online without installation using websites like:
  - ◆ <https://www.programiz.com/python-programming/online-compiler>
  - ◆ <https://replit.com/>

### 4. print() Function in Python

The print() function is used to display output on the screen.

### Basic Syntax

```
print("Hello, World!")
```

#### Output:

Hello, World!

### Printing Multiple Values

```
name = "Alice"
```

```
age = 25
```

```
print("Name:", name, "Age:", age)
```

#### Output:

Name: Alice Age: 25

### Using sep (Separator) Parameter

The sep parameter is used to specify a separator between values.

```
print("Apple", "Banana", "Cherry", sep=", ")
```

#### Output:

Apple, Banana, Cherry

### Using end Parameter

By default, print() moves to a new line after printing. You can change this using end.

```
print("Hello", end=" ")
```

```
print("World")
```

**Output:**

Hello World

**Printing with Formatting (f-strings)**

```
name = "Bob"  
age = 30  
print(f"My name is {name} and I am {age} years old.")
```

**Output:**

My name is Bob and I am 30 years old.

**Summary**

- ✓ Python is easy to install and run on any operating system.
- ✓ You can execute Python programs using interactive mode, script files, or online compilers.
- ✓ The print() function helps to display output and supports multiple arguments, separators, and formatting.

**Mini Project 1: Fancy Receipt Generator**

**Concepts Covered:**

- ✓ Using print() to format output
- ✓ Using escape sequences (\n, \t)
- ✓ Creating a structured receipt

**Problem Statement:**

Create a Python script that **prints a receipt** for a customer after a purchase.

**Sample Code:**

```
print("=" * 30)
print("\t\t SuperMart Receipt \t\t")
print("=" * 30)
```

```
# Items and prices
print("Item\tQty\tPrice")
print("-" * 30)
print("Apples\t2\t$1.50")
print("Bananas\t1\t$0.75")
print("Bread\t1\t$2.00")
print("Milk\t1\t$1.80")
```

```
print("-" * 30)
print("Total:\t\t\t $6.05")
print("=" * 30)
print("\tThank You! Visit Again ☺")
print("=" * 30)
```

**Example Output:**

```
=====
\t\t SuperMart Receipt \t\t
=====
Item      Qty   Price
-----
Apples      2     $1.50
Bananas     1     $0.75
Bread       1     $2.00
```

Milk      1      \$1.80

---

Total:      \$6.05

---

Thank You! Visit Again 😊

---

### What You Learn?

- Using \t for spacing
- Using \* for decorative borders
- Structuring output properly

### Mini Project 2: ASCII Art Banner

#### Concepts Covered:

✓ Using print() creatively

✓ ASCII art with print()

✓ Multiline string formatting

#### Problem Statement:

Create a script that prints a banner with ASCII art.

#### Sample Code:

```
print("====")
print("  ★ WELCOME TO PYTHON CAFE ☕  ")
print("====")
print("")
```

```
(\_)  
(o.o) Welcome!  
(>♥<) Enjoy your coffee ☕  
""")  
  
print("=====")  
print(" Special Offer: Buy 2 Get 1 Free! ☕")  
print("=====")
```

### Example Output:

```
=====  
★ WELCOME TO PYTHON CAFE ☕  
=====  
  
(\_)  
(o.o) Welcome!  
(>♥<) Enjoy your coffee ☕  
  
=====  
Special Offer: Buy 2 Get 1 Free! ☕  
=====
```

### What You Learn?

- Using `print()` for multiline ASCII art
- Using emoji for fun output
- Formatting banners for better appearance

## Day 22 Tasks

### Task 1: Print "Hello, World!"

Write a Python program that prints "Hello, World!" to the console.

### Task 2: Print Your Name

Write a program that prints your full name on the screen.

### Task 3: Print a Quote

Display your favorite quote inside double quotes.

#### Example:

"The only limit to our realization of tomorrow is our doubts of today." - Franklin D. Roosevelt

### Task 4: Print a Multi-line Message

Use triple quotes (""""") or \n to print a multi-line message.

#### Example:

Hello, Python Learner!

Welcome to this amazing journey.

Keep coding and have fun! 😊

### Task 5: Print a Simple Math Calculation

Write a program that prints the result of  $25 + 75 - 10$  using `print()`.

### Task 6: Print a Receipt Format

Create a simple receipt using `print()` and \t (tab spacing).

### Task 7: Print a Table

Use \t to create a simple multiplication table (e.g., 5 times table).

**Example:**

5 x 1 = 5

5 x 2 = 10

...

5 x 10 = 50

**Task 8:** Print an ASCII Art

Use print() to display a simple ASCII art of a smiley face or any shape.

**Example:**

(•\_•)

( >❤< )

**Task 9:** Print a Decorative Banner

Use print() with \* or = to print a banner with a message inside.

**Example:**

=====

Welcome to Python! 🎉

=====

**Task 10:** Print with Separators

Use the sep parameter in print() to separate words with a custom symbol.

**Example:**

```
print("Apple", "Banana", "Cherry", sep=" | ")
```

**Output:**

Apple | Banana | Cherry

### **Task 11: Print with end Parameter**

Use end to print two sentences in the same line.

#### **Example:**

```
print("Hello", end=" ")
print("World!")
```

#### **Output:**

Hello World!

### **Task 12: Print Using f-strings**

Use an f-string to format a message dynamically.

#### **Example:**

```
name = "Alice"
age = 25
print(f"My name is {name} and I am {age} years old.")
```

#### **Output:**

My name is Alice and I am 25 years old.

### **Task 13: Print a Countdown Timer**

Use multiple print() statements to display a countdown from 5 to 1, ending with "Go!".

#### **Example:**

```
5...
4...
3...
2...
1...
Go!
```

## Mini Project 1: Business Card Generator

### Concepts Covered:

- ✓ Using print() for formatting
- ✓ Escape sequences (\t, \n)
- ✓ Structuring output

### Task Description:

Write a Python script that prints a business card for a person with their name, job title, company, email, and phone number in a well-structured format.

### Expected Output Example:

```
=====
★ BUSINESS CARD
=====
Name: John Doe
Job Title: Software Engineer
Company: Tech Solutions Inc.
Email: johndoe@example.com
Phone: +1 234 567 8901
=====
```

## Mini Project 2: Movie Ticket Printout

### Concepts Covered:

- ✓ Using print() for structured formatting
- ✓ Escape sequences (\t, \n)
- ✓ Using decorative elements

### Task Description:

Create a Python script that prints a movie ticket format with details like movie name, showtime, seat number, and ticket price.

### Expected Output Example:

```
*****  
MOVIE TICKET   
*****  
Movie: Spider-Man: No Way Home  
Showtime: 7:30 PM  
Seat No: A12  
Price: $12.50  
*****  
Enjoy Your Movie!    
*****
```

### Bonus Challenge:

Modify these scripts to ask for user input (e.g., `input()`) and generate a custom business card or movie ticket dynamically!

## Day 23

### Variables and Data Types in Python

#### What is a Variable?

A variable is a name given to a value stored in memory. It acts as a container that holds data, which can be changed later in the program.

## Example of a Variable:

```
name = "Alice" # Storing a string value in a variable
age = 25      # Storing a number in a variable
```

Here, name and age are **variables** storing different types of data.

## What are Data Types?

A data type defines what kind of data a variable can hold. Python has several built-in data types.

## Common Data Types in Python

Data Type	Description	Example
int (Integer)	Whole numbers	x = 10
float (Floating Point)	Decimal numbers	y = 3.14
str (String)	Text values	name = "Alice"
bool (Boolean)	True/False values	is_valid = True
list (List)	Ordered collection	fruits = ["apple", "banana", "cherry"]
tuple (Tuple)	Immutable collection	coordinates = (10, 20)
dict (Dictionary)	Key-value pairs	person = {"name": "John", "age": 30}
set (Set)	Unordered unique values	unique_numbers = {1, 2, 3, 4}

## Easy Examples for Each Data Type

### **Integer (int)**

```
age = 25  
print(age) # Output: 25
```

### **Float (float)**

```
price = 99.99  
print(price) # Output: 99.99
```

### **String (str)**

```
greeting = "Hello, World!"  
print(greeting) # Output: Hello, World!
```

### **Boolean (bool)**

```
is_raining = False  
print(is_raining) # Output: False
```

### **List (list)**

```
fruits = ["apple", "banana", "cherry"]  
print(fruits[0]) # Output: apple
```

### **Tuple (tuple)**

```
coordinates = (10, 20)  
print(coordinates[1]) # Output: 20
```

### Dictionary (dict)

```
student = {"name": "Alice", "age": 21}  
print(student["name"]) # Output: Alice
```

### Set (set)

```
unique_numbers = {1, 2, 3, 4, 4, 2}  
print(unique_numbers) # Output: {1, 2, 3, 4}
```

### Summary

- ✓ Variables store data in memory.
- ✓ Python automatically assigns a data type based on the value.
- ✓ Data types define how data is stored and used in a program.

## Understanding input() and f-strings (f"""") in Python

### 1. input() – Getting User Input

The `input()` function allows users to enter data into a Python program. By default, it always returns a string.

#### Example: Simple User Input

```
name = input("Enter your name: ") # User enters: Alice  
print("Hello, " + name + "!")
```

#### Output (if user enters "Alice")

Enter your name: Alice  
Hello, Alice!

### **Example: Getting a Number as Input**

Since `input()` returns a string, we **must convert it** to an integer (`int`) or float (`float`) for calculations.

```
age = int(input("Enter your age: ")) # Convert input to integer
print("Next year, you will be", age + 1, "years old.")
```

#### **Output (if user enters 25):**

Enter your age: 25  
Next year, you will be 26 years old.

## **2. f-string (f""" – Formatting Strings Easily**

An f-string (formatted string) allows you to insert variables directly into a string using `{}`. It makes string formatting easier and more readable.

### **Example: Using f-string**

```
name = "Alice"
age = 25
print(f"My name is {name} and I am {age} years old.")
```

#### **Output:**

My name is Alice and I am 25 years old.

### **Example: f-string with Expressions**

You can also perform calculations inside `{}`.

```
price = 100
discount = 20
final_price = price - discount
print(f"The final price after discount is ${final_price}.")
```

**Output:**

The final price after discount is \$80.

**Combining input() with f-strings**

You can use f-strings with input() to display user input dynamically.

**Example: Personalized Greeting**

```
name = input("Enter your name: ")  
age = int(input("Enter your age: "))  
print(f"Hello {name}, you are {age} years old!")
```

**Output (if user enters "Bob" and "30")**

```
Enter your name: Bob  
Enter your age: 30  
Hello Bob, you are 30 years old!
```

**Summary**

- ✓ input() is used to get user input (always returns a string).
- ✓ Convert input() using int() or float() for calculations.
- ✓ f-strings (f"""") allow easy string formatting with {}.
- ✓ f-strings can include variables and expressions.

**Understanding type() in Python****What is type()?**

The type() function in Python is used to check the data type of a variable or value. It helps us understand what kind of data is being stored in a variable.

## Basic Syntax

```
type(variable_name)
```

### Example: Checking Data Types

```
x = 10  
y = 3.14  
z = "Hello"  
a = True
```

```
print(type(x)) # Output: <class 'int'>  
print(type(y)) # Output: <class 'float'>  
print(type(z)) # Output: <class 'str'>  
print(type(a)) # Output: <class 'bool'>
```

## Checking Data Types of User Input

Since `input()` always returns a string, you can use `type()` to verify it.

```
user_input = input("Enter something: ")  
print(type(user_input)) # Output: <class 'str'>
```

Even if the user enters a number (e.g., "10"), the output will still be `<class 'str'>` because `input()` returns everything as a string.

To convert it into an integer:

```
num = int(input("Enter a number: "))  
print(type(num)) # Output: <class 'int'>
```

### Example: Checking Type in a List

```
data = [10, "hello", 3.5, True]
```

```
for item in data:  
    print(f"Value: {item}, Type: {type(item)}")
```

#### Output:

```
Value: 10, Type: <class 'int'>  
Value: hello, Type: <class 'str'>  
Value: 3.5, Type: <class 'float'>  
Value: True, Type: <class 'bool'>
```

### Summary

- ✓ type() helps identify the data type of a variable or value.
- ✓ input() always returns a string, so type() can confirm it.
- ✓ Useful for debugging and dynamic data handling.

### Mini Project 1: Simple Personal Info Formatter

Concepts Used: print(), variables, datatypes, input(), type(), f-string

#### Description:

A program that asks for the user's name, age, and height, then displays the details in a well-formatted manner.

#### Code:

```
# Taking user input  
name = input("Enter your name: ")  
age = int(input("Enter your age: "))
```

```
height = float(input("Enter your height in cm: "))

# Checking types
print(f"Type of name: {type(name)}") # str
print(f"Type of age: {type(age)}") # int
print(f"Type of height: {type(height)}") # float

# Printing formatted user details
print("\n===== Personal Information =====")
print(f"Hello {name}, you are {age} years old and {height} cm tall.")
```

**Sample Output:**

```
Enter your name: Alice
Enter your age: 25
Enter your height in cm: 162.5
Type of name: <class 'str'>
Type of age: <class 'int'>
Type of height: <class 'float'>
```

```
===== Personal Information =====
Hello Alice, you are 25 years old and 162.5 cm tall.
```

**Mini Project 2: Simple Bill Calculator**

Concepts Used: print(), variables, datatypes, input(), type(), f-string

**Description:**

A program that asks for an item's price and quantity, then calculates the total bill with tax and formats the output nicely.

**Code:**

```

# Taking user input
item_name = input("Enter the item name: ")
price = float(input("Enter the price per item: "))
quantity = int(input("Enter the quantity: "))

# Calculating total bill (including 10% tax)
subtotal = price * quantity
tax = subtotal * 0.10
total = subtotal + tax

# Checking data types
print(f"Type of item_name: {type(item_name)}") # str
print(f"Type of price: {type(price)}") # float
print(f"Type of quantity: {type(quantity)}") # int
print(f"Type of total: {type(total)}") # float

# Printing formatted bill
print("\n===== Bill Summary =====")
print(f"Item: {item_name}")
print(f"Price per item: ${price:.2f}")
print(f"Quantity: {quantity}")
print(f"Subtotal: ${subtotal:.2f}")
print(f"Tax (10%): ${tax:.2f}")
print(f"Total Amount: ${total:.2f}")

```

**Sample Output:**

```

Enter the item name: Laptop
Enter the price per item: 750.50
Enter the quantity: 2
Type of item_name: <class 'str'>

```

Type of price: <class 'float'>

Type of quantity: <class 'int'>

Type of total: <class 'float'>

===== Bill Summary =====

Item: Laptop

Price per item: \$750.50

Quantity: 2

Subtotal: \$1501.00

Tax (10%): \$150.10

Total Amount: \$1651.10

### **Summary**

Project 1: Formats personal information using user input and type()

Project 2: Calculates the total bill with tax and displays the formatted output using f-strings

## **Day 23 Tasks**

### **1. Print Your Name**

Write a program that prints your full name using print().

#### **Example Output:**

My name is John Doe.

### **2. Print Multiple Lines**

Use print() to display a welcome message on multiple lines.

**Example Output:**

Welcome to Python!  
This is a beginner-friendly language.  
Let's start coding.

**3. Variable Assignment and Printing**

Create variables for your name, age, and favorite color. Print them.

**Example Output:**

Name: Alice  
Age: 25  
Favorite Color: Blue

**4. Data Type Identification**

Define different types of variables (string, integer, float, boolean) and print their types using type().

**Example Output:**

```
<class 'str'>
<class 'int'>
<class 'float'>
<class 'bool'>
```

**5. Taking User Input**

Ask the user to enter their name and print a welcome message.

**Example Input:**

Enter your name: John

**Example Output:**

Hello, John! Welcome to Python.

## 6. Sum of Two Numbers

Take two numbers as input, convert them to integers, add them, and display the result.

### **Example Input:**

Enter first number: 10

Enter second number: 20

### **Example Output:**

The sum is: 30

## 7. Data Type Conversion

Ask the user to enter a number, print its type, convert it to a float, and print its new type.

### **Example Input:**

Enter a number: 5

### **Example Output:**

Original type: <class 'str'>

Converted type: <class 'float'>

## 8. Formatted Sentence Using f-strings

Ask the user for their name, age, and city, and display a sentence using an f-string.

### **Example Input:**

Enter your name: Alice

Enter your age: 30

Enter your city: London

### **Example Output:**

Hello Alice! You are 30 years old and live in London.

## 9. Area of a Rectangle

Take the length and width of a rectangle as input and calculate the area using an f-string.

### **Example Input:**

Enter length: 5

Enter width: 10

### **Example Output:**

The area of the rectangle is: 50 square units.

## 10. Printing a Receipt

Ask the user for an item name, quantity, and price, then display a formatted bill using f-strings.

### **Example Input:**

Enter item name: Book

Enter quantity: 3

Enter price per item: 10

### **Example Output:**

Item: Book

Quantity: 3

Price per item: \$10.00

Total cost: \$30.00

## 11. Swap Two Variables

Take two numbers from the user and swap them without using a third variable.

### **Example Input:**

Enter first number: 5

Enter second number: 10

**Example Output:**

After swapping:

First number: 10

Second number: 5

**12. Temperature Converter**

Take a temperature in Celsius as input, convert it to Fahrenheit, and display it using an f-string.

**Formula:**

$$\text{Fahrenheit} = (\text{Celsius} * 9/5) + 32$$

**Example Input:**

Enter temperature in Celsius: 30

**Example Output:**

30°C is equal to 86°F.

**13. Simple Profile Display**

Take the user's name, age, height, and favorite hobby, then display a formatted profile.

**Example Input:**

Enter your name: John

Enter your age: 28

Enter your height (in cm): 175

Enter your favorite hobby: Reading

**Example Output:**

===== User Profile =====

Name: John

Age: 28 years old

Height: 175 cm

Hobby: Reading

=====

### **Mini Project 1: Simple Student Report Card**

Concepts Used: print(), variables, datatypes, input(), type(), f-string

#### **Description:**

This project will take a student's name, class, marks in three subjects, calculate the total and percentage, and display a formatted report card.

#### **¶ Sample Output:**

Enter student name: Alice

Enter class: 8

Enter Math marks: 85

Enter Science marks: 90

Enter English marks: 80

Type of student\_name: <class 'str'>

Type of student\_class: <class 'str'>

Type of math\_marks: <class 'int'>

Type of percentage: <class 'float'>

===== Student Report Card =====

Name : Alice

Class : 8

Math : 85

Science : 90

English : 80

Total Marks: 255

Percentage : 85.00%

=====

## **Mini Project 2: Employee Salary Calculator**

Concepts Used: print(), variables, datatypes, input(), type(), f-string

### **Description:**

This project will take an employee's name, basic salary, and allowances, calculate deductions (tax), and display the final salary slip.

#### **Sample Output:**

Enter employee name: John Doe

Enter basic salary: 50000

Enter total allowances: 10000

Type of employee\_name: <class 'str'>

Type of basic\_salary: <class 'float'>

Type of net\_salary: <class 'float'>

**===== Salary Slip =====**

Employee Name : John Doe

Basic Salary : \$50000.00

Allowances : \$10000.00

Tax Deduction : \$5000.00

Net Salary : \$55000.00

=====

## Day 24

### Operators in python

#### What are Operators?

Operators are symbols that perform operations on variables and values. Python has different types of operators for arithmetic, comparison, logical operations, etc.

#### Types of Operators in Python

##### 1. Arithmetic Operators (+, -, \*, /, //, %, )

Used to perform mathematical operations.

Operator	Description	Example
+	Addition	$5 + 3 \rightarrow 8$
-	Subtraction	$5 - 3 \rightarrow 2$
*	Multiplication	$5 * 3 \rightarrow 15$
/	Division (float)	$5 / 2 \rightarrow 2.5$

//	Floor Division	$5 // 2 \rightarrow 2$
%	Modulus (Remainder)	$5 \% 2 \rightarrow 1$
**	Exponentiation	$2 ** 3 \rightarrow 8$

**Example:**

```
a = 10
b = 3
print(f"Addition: {a + b}") # 13
print(f"Subtraction: {a - b}") # 7
print(f"Multiplication: {a * b}") # 30
print(f"Division: {a / b}") # 3.3333
print(f"Floor Division: {a // b}") # 3
print(f"Modulus: {a % b}") # 1
print(f"Exponentiation: {a ** b}") # 1000
```

**2. Comparison Operators (==, !=, >, <, >=, <=)**

Used to compare two values and return True or False.

Operator	Description	Example
==	Equal to	$5 == 5 \rightarrow \text{True}$
!=	Not equal to	$5 != 3 \rightarrow \text{True}$
>	Greater than	$5 > 3 \rightarrow \text{True}$
<	Less than	$5 < 3 \rightarrow \text{False}$
>=	Greater than or equal to	$5 >= 5 \rightarrow \text{True}$
<=	Less than or equal to	$5 <= 3 \rightarrow \text{False}$

**Example:**

```
x = 10
y = 20
print(x == y) # False
```

```
print(x != y) # True
print(x > y) # False
print(x < y) # True
print(x >= y) # False
print(x <= y) # True
```

### 3. Logical Operators (and, or, not)

Used to combine conditional statements.

Operator	Description	Example
and	Returns True if both conditions are true	(5 > 2 and 10 > 5) → True
or	Returns True if at least one condition is true	(5 > 10 or 10 > 5) → True
not	Reverses the result	not(5 > 2) → False

**Example:**

```
a = 5
b = 10
print(a > 2 and b > 5) # True
print(a > 10 or b > 5) # True
print(not(a > 2)) # False
```

### 4. Assignment Operators (=, +=, -=, \*=, /=, //=, %=, =)

Used to assign values to variables.

Operator	Example	Equivalent To
=	x = 5	x = 5
+=	x += 3	x = x + 3

<code>-=</code>	<code>x -= 3</code>	<code>x = x - 3</code>
<code>*=</code>	<code>x *= 3</code>	<code>x = x * 3</code>
<code>/=</code>	<code>x /= 3</code>	<code>x = x / 3</code>
<code>//=</code>	<code>x //= 3</code>	<code>x = x // 3</code>
<code>%=</code>	<code>x %= 3</code>	<code>x = x % 3</code>
<code>**=</code>	<code>x **= 3</code>	<code>x = x ** 3</code>

**Example:**

```

x = 10
x += 5 # x = x + 5 → 15
print(x)
x *= 2 # x = x * 2 → 30
print(x)

```

**5. Identity Operators (is, is not)**

Used to compare memory locations of two objects.

Operator	Description	Example
<code>is</code>	Returns True if both variables refer to the same object	<code>x is y</code>
<code>is not</code>	Returns True if both variables do not refer to the same object	<code>x is not y</code>

**Example:**

```

a = [1, 2, 3]
b = a
c = [1, 2, 3]
print(a is b) # True (same memory location)
print(a is c) # False (different memory location)
print(a is not c) # True

```

## 6. Membership Operators (in, not in)

Used to check if a value exists in a sequence (list, tuple, string).

Operator	Description	Example
in	Returns True if a value exists in the sequence	"a" in "apple" → True
not in	Returns True if a value does not exist in the sequence	"x" not in "apple" → True

### Example:

```
text = "Hello, Python!"
print("Python" in text) # True
print("Java" not in text) # True
```

## 7. Bitwise Operators (&, |, ^, ~, <<, >>)

Used to perform bit-level operations.

Operator	Description	Example
&	AND	5 & 3 → 1
'`	OR	
^	XOR	5 ^ 3 → 6
~	NOT	~5 → -6
<<	Left shift	5 << 1 → 10
>>	Right shift	5 >> 1 → 2

### Example:

```
a = 5 # 0101
b = 3 # 0011
print(a & b) # 0001 → 1
print(a | b) # 0111 → 7
print(a ^ b) # 0110 → 6
```

```
print(~a) # -6
print(a << 1) # 1010 → 10
print(a >> 1) # 0010 → 2
```

## Summary Table

Type	Example
Arithmetic	+, -, *, /, //, %, **
Comparison	==, !=, >, <, >=, <=
Logical	and, or, not
Assignment	=, +=, -=, *=, /=, //=, %=, **=
Identity	is, is not
Membership	in, not in
Bitwise	&,

## Conclusion

- Operators help perform different operations on variables and values.
- Arithmetic, comparison, logical, assignment, identity, membership, and bitwise operators are commonly used in Python.
- These operators increase the efficiency of coding and help manipulate data easily.

## Conditional Statements

### What are Conditional Statements?

Conditional statements allow a program to make decisions and execute specific code based on conditions. They help control the flow of a program.

### **Example Scenario:**

- If it is raining, take an umbrella.
- If you score more than 50, you pass; otherwise, you fail.

In Python, we use if, elif, and else to write conditional statements.

## **Types of Conditional Statements in Python**

### **1. if Statement**

The if statement checks a condition and executes the code inside **only if the condition is True**.

#### **Syntax:**

```
if condition:  
    # Code to execute if condition is True
```

#### **Example:**

```
age = 18  
if age >= 18:  
    print("You are eligible to vote!")
```

#### **Output:**

You are eligible to vote!

### **2. if-else Statement**

The else block runs when the if condition is False.

#### **Syntax:**

```
if condition:  
    # Code if condition is True
```

```
else:  
    # Code if condition is False
```

**Example:**

```
age = 16  
if age >= 18:  
    print("You can vote!")  
else:  
    print("You cannot vote.")
```

**Output:**

You cannot vote.

**3. if-elif-else Statement**

When multiple conditions need to be checked, we use elif.

**Syntax:**

```
if condition1:  
    # Code if condition1 is True  
elif condition2:  
    # Code if condition2 is True  
else:  
    # Code if all conditions are False
```

**Example:**

marks = 85

```
if marks >= 90:  
    print("Grade: A")  
elif marks >= 80:  
    print("Grade: B")
```

```
elif marks >= 70:  
    print("Grade: C")  
else:  
    print("Grade: D")
```

**Output:**

Grade: B

#### 4. Nested if Statement

An if statement inside another if statement is called a **nested if**.

**Syntax:**

```
if condition1:  
    if condition2:  
        # Code if both conditions are True
```

**Example:**

```
age = 20  
has_id = True
```

```
if age >= 18:  
    if has_id:  
        print("You can enter the club!")  
    else:  
        print("You need an ID to enter.")  
else:  
    print("You are not allowed.")
```

**Output:**

You can enter the club!

## Summary Table

Conditional Statement	Description
if	Executes code if condition is True
if-else	Runs one block if True, another if False
if-elif-else	Checks multiple conditions
Nested if	if inside another if

## Conclusion

- Conditional statements control the flow of execution based on conditions.
- if, if-else, if-elif-else, and nested if help make decisions in Python programs.
- These concepts are essential for writing dynamic and interactive programs.

## Mini Projects for Operators & Conditional Statements

### Mini Project: Simple Calculator

Concepts Used: Arithmetic Operators, Conditional Statements, input()

#### Description:

Create a simple calculator that takes two numbers and an operator (+, -, \*, /, %) from the user and performs the operation.

#### Code:

```
# Simple Calculator

# Taking user input
num1 = float(input("Enter first number: "))
operator = input("Enter an operator (+, -, *, /, %): ")
num2 = float(input("Enter second number: "))
# Performing calculation based on operator
if operator == "+":
```

```

result = num1 + num2
elif operator == "-":
    result = num1 - num2
elif operator == "*":
    result = num1 * num2
elif operator == "/":
    if num2 != 0: # Checking for division by zero
        result = num1 / num2
    else:
        result = "Error! Division by zero."
elif operator == "%":
    result = num1 % num2
else:
    result = "Invalid operator!"

print(f"Result: {result}")

```

**Example Output:**

Enter first number: 10  
 Enter an operator (+, -, \*, /, %): \*  
 Enter second number: 5  
 Result: 50.0

**Mini Project: Even or Odd Number Game**

Concepts Used: Modulus Operator (%), Conditional Statements, input()

**Description:**

Ask the user to enter a number, then check whether the number is even or odd.

**Code:**

```
# Even or Odd Number Checker

# Taking user input
num = int(input("Enter a number: "))

# Checking if the number is even or odd
if num % 2 == 0:
    print(f"{num} is an Even number.")
else:
    print(f"{num} is an Odd number.)
```

**Example Output:**

Enter a number: 7  
7 is an Odd number.

**Summary**

- Simple Calculator – Uses arithmetic operators and conditions to perform basic math operations.
- Even or Odd Checker – Uses modulus operator and conditional statements to determine if a number is even or odd.

**Day 24 Tasks****1. Arithmetic Operations**

- Write a program that takes two numbers as input and prints their sum, difference, product, quotient, and remainder.

**2. Compare Two Numbers**

- Write a program that asks the user for two numbers and prints whether the first number is greater than, less than, or equal to the second number.

### 3. Swap Two Numbers Without a Temporary Variable

- Swap two numbers using arithmetic operators (addition and subtraction or multiplication and division).

### 4. Check Leap Year

- Take a year as input and use the modulus (%) operator to check if it is a leap year.
- A leap year is divisible by 4, but if it's also divisible by 100, it must be divisible by 400.

### 5. Find the Largest of Three Numbers

- Ask the user for three numbers and determine which is the largest using comparison operators.

### 6. Calculate the Area of a Circle

- Take the radius as input and calculate the area using the formula:  $\text{area} = \pi * r^2$  (Use 3.14 for  $\pi$ )

### 7. Check Positive, Negative, or Zero

- Write a program that takes a number as input and checks if it is positive, negative, or zero using conditional statements.

### 8. Check Voting Eligibility

- Ask the user for their age and print whether they are eligible to vote (age  $\geq 18$ ).

## 9. Grade Calculator

- Take marks as input and assign grades based on these conditions:  
90+ → A  
80-89 → B  
70-79 → C  
60-69 → D  
Below 60 → Fail

## 10. Simple ATM Withdrawal Program

- Set an initial account balance (e.g., 5000).
- Ask the user how much they want to withdraw.
- If the amount is greater than the balance, print "Insufficient funds."  
Otherwise, subtract the amount and print the remaining balance.

## 11. Check Divisibility by 5 and 11

- Write a program that asks for a number and checks if it is divisible by both 5 and 11.

## 12. Check if a Character is a Vowel or Consonant

- Take a single character input and determine if it is a vowel (a, e, i, o, u) or a consonant.

## 13. Even or Odd Using Conditional Expressions

- Rewrite the even/odd program using a ternary (single-line) if-else statement:  
`num = int(input("Enter a number: "))  
print("Even" if num % 2 == 0 else "Odd")`

## Summary

These 13 tasks cover arithmetic, comparison, and logical operators along with if-else conditions to build decision-based programs.

## Mini Project: Discount Calculator for a Shopping Store

Concepts Used: Arithmetic Operators, Conditional Statements, input()

### Description:

Create a program that asks the user for the total bill amount and applies a discount based on the following conditions:

### Discount Rules:

- If the bill is ₹5000 or more, give a 20% discount.
- If the bill is between ₹3000 and ₹4999, give a 10% discount.
- If the bill is between ₹1000 and ₹2999, give a 5% discount.
- If the bill is less than ₹1000, no discount.

### Example Output:

Enter your total bill amount: ₹3500

Discount Applied: ₹350.00

Final Bill Amount: ₹3150.00

## Mini Project: Rock, Paper, Scissors Game

Statements, input()

### Description:

Create a Rock, Paper, Scissors game where the user plays against the computer. The computer randomly selects Rock, Paper, or Scissors, and the user inputs their choice. The winner is determined using these rules:

- Rock beats Scissors
- Scissors beats Paper
- Paper beats Rock
- If both choices are the same, it's a tie!

#### **Example Output:**

Enter your choice (rock, paper, scissors): rock

Computer chose: scissors

You Win!

## **Day 25**

### **Loops in Python**

Loops are used to repeat a block of code multiple times until a condition is met. Instead of writing the same code repeatedly, loops help in automating repetitive tasks.

Python has three types of loops:

1. For Loop
2. While Loop
3. Nested Loop (Loop inside another loop)

### **For Loop**

A for loop is used when you want to iterate over a sequence (list, tuple, string, etc.).

### Syntax:

```
for variable in sequence:  
    # Code to execute in each iteration
```

### Example 1: Print each item in a list

```
fruits = ["apple", "banana", "cherry"]  
for fruit in fruits:  
    print(fruit)
```

### Output:

```
apple  
banana  
cherry
```

### Example 2: Using range() in a for loop

```
for i in range(1, 6): # Prints numbers from 1 to 5  
    print(i)
```

### Output:

```
1  
2  
3  
4  
5
```

## While Loop

A while loop runs as long as the condition is True.

**Syntax:**

```
while condition:
    # Code to execute
```

**Example 1: Print numbers from 1 to 5 using while loop**

```
i = 1
while i <= 5:
    print(i)
    i += 1 # Increment `i`
```

**Output:**

```
1
2
3
4
5
```

**Example 2: Keep asking for input until the user enters "exit"**

```
user_input = ""
while user_input != "exit":
    user_input = input("Enter a word (or type 'exit' to stop): ")
    print("You entered:", user_input)
```

**Nested Loops**

A loop inside another loop is called a **nested loop**.

**Example: Print a pattern using nested loops**

```
for i in range(1, 4): # Outer loop (Runs 3 times)
    for j in range(1, 4): # Inner loop (Runs 3 times for each outer loop)
        print(i, j)
```

**Output:**

```

1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3

```

**Summary:**

<b>Loop Type</b>	<b>When to Use</b>
For Loop	When you know the number of iterations (e.g., iterating through a list or range)
While Loop	When the number of iterations is unknown (e.g., looping until a condition is met)
Nested Loop	When working with multi-level structures like matrices, patterns, or complex loops

**Python For Loop - Advanced Concepts****Python For Loop with String**

A for loop can be used to iterate through each character of a string.

**Syntax:**

```
for char in string:
    # Code to execute
```

### **Example: Iterate over a string**

```
text = "Python"  
for char in text:  
    print(char)
```

#### **Output:**

```
P  
y  
t  
h  
o  
n
```

### **Example: Count vowels in a string**

```
text = "Hello World"  
vowel_count = 0  
for char in text:  
    if char.lower() in "aeiou":  
        vowel_count += 1  
print("Number of vowels:", vowel_count)
```

#### **Output:**

Number of vowels: 3

### **Using range() with For Loop**

The range() function generates a sequence of numbers. It is commonly used in loops.

**Syntax:**

- `for i in range(start, stop, step):`  
    # Code to execute
- `start` → (Optional) Starting value (default = 0)
- `stop` → Ending value (not included)
- `step` → (Optional) Step value (default = 1)

**Example: Print numbers from 1 to 5**

```
for i in range(1, 6):  
    print(i)
```

**Output:**

```
1  
2  
3  
4  
5
```

**Example: Print even numbers from 2 to 10**

```
for i in range(2, 11, 2): # Step of 2  
    print(i)
```

**Output:**

```
2  
4  
6  
8  
10
```

## Control Statements: continue, break, pass, else

Control statements modify the loop behavior.

### **break: Stop the loop completely**

```
for i in range(1, 6):
    if i == 3:
        break # Stop the loop when i = 3
    print(i)
```

#### **Output:**

```
1
2
```

### **continue: Skip the current iteration**

```
for i in range(1, 6):
    if i == 3:
        continue # Skip when i = 3
    print(i)
```

#### **Output:**

```
1
2
4
5
```

### **pass: Do nothing (placeholder)**

```
for i in range(1, 6):
    if i == 3:
        pass # Placeholder for future code
    print(i)
```

**Output:**

```
1  
2  
3  
4  
5
```

**else with for loop (Runs if loop completes normally)**

```
for i in range(1, 4):  
    print(i)  
else:  
    print("Loop completed successfully!")
```

**Output:**

```
1  
2  
3  
Loop completed successfully!
```

**Using enumerate() with For Loop**

enumerate() gives both index and value when iterating.

**Syntax:**

```
for index, value in enumerate(iterable, start=0):  
    # Code to execute
```

**Example: Display index and character**

```
text = "Python"  
for index, char in enumerate(text):  
    print(f"Index {index}: {char}")
```

**Output:**

Index 0: P  
Index 1: y  
Index 2: t  
Index 3: h  
Index 4: o  
Index 5: n

**Example: Print list with index**

```
fruits = ["Apple", "Banana", "Cherry"]
for index, fruit in enumerate(fruits, start=1):
    print(f"{index}. {fruit}")
```

**Output:**

1. Apple  
2. Banana  
3. Cherry

**Nested For Loops**

A loop inside another loop is a nested loop.

**Example: Multiplication Table**

```
for i in range(1, 4): # Outer loop
    for j in range(1, 4): # Inner loop
        print(f"{i} x {j} = {i * j}")
    print("----")
```

**Output:**

1 x 1 = 1  
1 x 2 = 2  
1 x 3 = 3

----

$2 \times 1 = 2$

$2 \times 2 = 4$

$2 \times 3 = 6$

----

$3 \times 1 = 3$

$3 \times 2 = 6$

$3 \times 3 = 9$

----

### Example: Print a pattern

```
for i in range(1, 6):
    for j in range(i):
        print("*", end=" ")
    print()
```

### Output:

```
*
```

 $* *$ 
 $* * *$ 
 $* * * *$ 
 $* * * * *$ 

### Summary Table

Concept	Description	Example
For Loop with String	Iterate over each character in a string	"Python" → P, y, t, h, o, n
Using range() with For Loop	Iterate over numbers in a range	range(1, 6) → 1, 2, 3, 4, 5
break	Exit the loop early	Stops at $i == 3$

continue	Skip current iteration	Skips <code>i == 3</code>
pass	Do nothing (placeholder)	Useful for future code
else with for	Runs if the loop finishes normally	"Loop completed successfully!"
enumerate()	Get both index and value	"Python" → (0, 'P'), (1, 'y')...
Nested For Loop	Loop inside another loop	Print patterns, tables

## Mini Project 1: Student Grades Analyzer

### Concepts Used:

- ✓ For Loop with String
- ✓ Using range() with For Loop
- ✓ Control Statements (break, continue, else)
- ✓ Using enumerate() with for loop

### Project Description:

Create a Python program that takes student names and their marks as input, calculates the average marks, and identifies top-performing students.

### Expected Features:

- ✓ Input student names and marks
- ✓ Use enumerate() to display student rank
- ✓ Use range() to process marks
- ✓ Identify top performers (marks ≥ 80)
- ✓ Use continue to skip students with absent marks (-1)
- ✓ Use break if the user enters "stop"

**Code Implementation:**

```
students = {  
    "John": 85,  
    "Emma": 92,  
    "Liam": 78,  
    "Olivia": -1, # Absent  
    "Sophia": 90,  
    "James": 65  
}  
  
total_marks = 0  
count = 0  
  
print("Student Rankings:")  
for index, (name, marks) in enumerate(students.items(), start=1):  
  
    if marks == -1:  
        continue # Skip absent students  
  
    total_marks += marks  
    count += 1  
  
    print(f"{index}. {name} - {marks} marks")  
  
avg_marks = total_marks / count if count > 0 else 0  
print(f"\nClass Average Marks: {avg_marks}")  
  
# Identify top performers  
print("\nTop Performers:")  
for name, marks in students.items():  
    if marks >= 80:
```

```
print(f"🏆 {name} - {marks} marks")
```

**Sample Output:**

Student Rankings:

1. John - 85 marks
2. Emma - 92 marks
3. Liam - 78 marks
4. Sophia - 90 marks
5. James - 65 marks

Class Average Marks: 82.0

Top Performers:

- 🏆 John - 85 marks
- 🏆 Emma - 92 marks
- 🏆 Sophia - 90 marks

## Mini Project 2: Number Pyramid Generator

**Concepts Used:**

- ✓ Using range() with For Loop
- ✓ Control Statements (break, continue)
- ✓ Nested For Loops

**Project Description:**

Create a Number Pyramid Generator where the user inputs the number of rows, and the program generates a pyramid pattern. If the user enters a negative number, the program should stop.

## Expected Features:

- ✓ Ask for user input (number of rows)
- ✓ Use nested loops to print a pyramid
- ✓ Stop execution if input is negative (break)
- ✓ Use continue to skip even rows

## Code Implementation:

```
rows = int(input("Enter number of rows for the pyramid: "))
```

```
if rows < 0:  
    print("Invalid input. Exiting...")  
else:  
    for i in range(1, rows + 1):  
        if i % 2 == 0:  
            continue # Skip even rows  
        for j in range(i):  
            print(i, end=" ")  
    print()
```

## Sample Output:

```
Enter number of rows for the pyramid: 5
```

```
1  
3 3 3  
5 5 5 5 5
```

## Day 25 Tasks

1. Write a program to print each character of the string "PYTHON" using a for loop.
2. Create a program that counts the number of vowels in a given string.
3. Write a program to reverse a string using a for loop.
4. Write a program to print numbers from 1 to 20 using range().
5. Create a program that prints only even numbers from 2 to 50 using range().
6. Write a program to print numbers in reverse order from 10 to 1 using range().
7. Write a program that asks the user to enter numbers until they enter 0, then stop the loop using break.
8. Create a loop that skips multiples of 5 from 1 to 50 using continue.
9. Write a program where a for loop iterates through numbers from 1 to 10, and if the number is 5, use pass to do nothing.
10. Write a program that iterates through numbers from 1 to 10, and after the loop ends, print "Loop finished successfully" using the else block.
11. Create a program that prints each character of "HELLO" with its index position using enumerate().
12. Write a program that asks the user for a sentence and prints each word with its position number using enumerate().
13. Write a program to print a multiplication table (1 to 10) using a nested loop.

## Mini Project 1: Word Frequency Counter

Concepts Used: Python For Loop with String, Using enumerate(), range(), Control Statements

**Objective:**

Create a program that counts the frequency of each word in a given sentence and displays the result in a neat format.

**Steps to Implement:**

1. Take a sentence as input from the user.
2. Convert the sentence into a list of words using `.split()`.
3. Use a for loop to iterate through each word.
4. Use a dictionary to store the frequency of each word.
5. Print the word along with its frequency using `enumerate()`

**Sample Output:**

Enter a sentence: python is fun and python is easy

Word Frequency Count:

1. python -> 2
2. is -> 2
3. fun -> 1
4. and -> 1
5. easy -> 1

**Mini Project 2: Triangle Pattern Generator**

Concepts Used: Nested Loops, `range()`, Control Statements (`break`, `continue`)

**Objective:**

Create a Triangle Pattern Generator that takes a number input from the user and prints a triangle pattern.

**Steps to Implement:**

1. Take an integer input from the user (number of rows).

2. Use a nested for loop to print the triangle pattern.
3. Use break to stop printing at a specific row if the user enters an odd number.
4. Use continue to skip even numbers in printing.

**Sample Output:**

Enter the number of rows: 7

Triangle Pattern:

```
*  
* * *  
* * * * *  
* * * * * *
```

Triangle generation completed.

## Day 26

### While Loop in Python

#### What is a While Loop?

A while loop in Python is used to execute a block of code as long as a condition is True.

#### Syntax of while loop:

while condition:

```
# Code to execute
```

**Example: Printing numbers from 1 to 5 using a while loop:**

```
count = 1
while count <= 5:
    print(count)
    count += 1
```

**Output:**

```
1
2
3
4
5
```

**Infinite While Loop in Python**

An infinite loop occurs when the condition in the while statement never becomes False. This results in an endless execution.

**Example of Infinite Loop:**

```
while True:
    print("This is an infinite loop!")
```

**Note:** To stop an infinite loop in Python, press Ctrl + C.

**While Loop with continue Statement**

The continue statement skips the remaining code in the current iteration and moves to the next iteration.

**Example: Printing odd numbers from 1 to 10 using continue:**

```
num = 0
while num < 10:
    num += 1
```

```
if num % 2 == 0:  
    continue # Skip even numbers  
    print(num)
```

**Output:**

```
1  
3  
5  
7  
9
```

**While Loop with break Statement**

The break statement is used to exit the loop immediately, even if the condition is still True.

**Example: Stopping the loop when number reaches 5:**

```
num = 1  
while num <= 10:  
    print(num)  
    if num == 5:  
        break # Stop loop when num is 5  
    num += 1
```

**Output:**

```
1  
2  
3  
4  
5
```

## While Loop with pass Statement

The pass statement is used when you need a placeholder for future code. It does nothing and allows the program to run without errors.

### Example:

```
x = 1
while x <= 5:
    if x == 3:
        pass # Placeholder, does nothing
    else:
        print(x)
    x += 1
```

### Output:

```
1
2
4
5
```

(The number **3** is skipped because of pass, but no effect on loop execution.)

## While Loop with else Statement

The else statement in a while loop executes only if the loop runs completely without hitting a break statement.

### Example:

```
num = 1
while num <= 3:
    print(num)
    num += 1
else:
    print("Loop finished successfully!")
```

**Output:**

1  
2  
3

Loop finished successfully!

**Mini Project 1: ATM PIN Verification System**

Concepts Used: while loop, break, continue, else

**Project Overview:**

- The user is asked to enter a 4-digit PIN.
- They get 3 attempts to enter the correct PIN.
- If the PIN is correct, access is granted.
- If the PIN is incorrect 3 times, access is denied.

**Code Implementation:**

```
correct_pin = "1234" # Set a correct PIN
attempts = 3

while attempts > 0:
    pin = input("Enter your 4-digit PIN: ")

    if pin == correct_pin:
        print("✓ Access Granted!")
        break # Exit the loop if PIN is correct
    else:
        attempts -= 1
        print(f"✗ Incorrect PIN! {attempts} attempts left.")

    if attempts == 0:
```

```
    print("∅ Access Denied. Your account is locked.")  
else:  
    continue # Skip to the next attempt
```

### **Output Example:**

```
Enter your 4-digit PIN: 1111  
✗ Incorrect PIN! 2 attempts left.  
Enter your 4-digit PIN: 2222  
✗ Incorrect PIN! 1 attempts left.  
Enter your 4-digit PIN: 1234  
✓ Access Granted!
```

## **Mini Project 2: Number Guessing Game**

Concepts Used: while loop, break, else, pass

### **Project Overview:**

- The program randomly selects a number between 1 and 20.
- The user gets 5 attempts to guess the number.
- If they guess correctly, they win.
- If they fail, the correct number is revealed.

### **Code Implementation:**

```
import random  
  
secret_number = random.randint(1, 20)  
attempts = 5  
  
while attempts > 0:  
    guess = int(input("Guess a number between 1 and 20: "))
```

```
if guess == secret_number:  
    print("🎉 Congratulations! You guessed the correct number!")  
    break # Exit the loop when guessed correctly  
elif guess > secret_number:  
    print("🔴 Too high! Try again.")  
else:  
    print("🟡 Too low! Try again.")  
  
attempts -= 1  
  
if attempts == 0:  
    print(f"❌ Out of attempts! The correct number was {secret_number}.")  
else:  
    pass # Placeholder for future enhancements
```

### Output Example:

Guess a number between 1 and 20: 10  
🟡 Too low! Try again.  
Guess a number between 1 and 20: 15  
🔴 Too high! Try again.  
Guess a number between 1 and 20: 13  
🎉 Congratulations! You guessed the correct number!

### Day 26 Tasks

Task 1: Create an infinite loop that prints "Hello, World!" continuously.

Task 2: Modify the infinite loop to stop printing after 5 seconds.

Task 3: Write an infinite loop that asks the user for input and prints it back, breaking when "exit" is entered.

Task 4: Print all even numbers from 1 to 20 using while and continue.

Task 5: Ask the user to enter a number. If the number is negative, ignore it and ask again. Stop only when a positive number is entered.

Task 6: Write a while loop that prints all numbers except multiples of 3 between 1 and 30.

Task 7: Write a number guessing game where the user must guess a number between 1 and 10. Stop the game when the correct number is guessed.

Task 8: Create a while loop that keeps asking for the password. Break when the correct password is entered.

Task 9: Simulate a simple ATM system where the user gets 3 attempts to enter the correct PIN. If they fail, display "Account Locked" and exit.

Task 10: Write a loop that iterates 10 times but does nothing inside the loop using pass.

Task 11: Use pass in a loop that will later be implemented but currently does nothing.

Task 12: Create a while loop that prints numbers from 1 to 5. If the loop completes naturally (without break), print "Loop completed successfully" using else.

Task 13: Write a while loop that asks the user for a word. If they enter "Python", break the loop. Otherwise, if the loop finishes without "Python", print "You never entered 'Python'!" using else.

## Mini Project 1: To-Do List Manager (Using while, continue, break, else)

### Task:

Create a To-Do List Manager where users can:

- Add tasks (User enters a task to add).
- View tasks (Display all tasks).
- Remove tasks (User enters the task number to delete it).
- Exit the program when the user types "exit".

### Conditions to Use:

- ✓ Use an infinite while loop to keep asking for actions.
- ✓ If the user enters an empty task, use continue to ignore and ask again.
- ✓ If the user types "exit", use break to stop the program.
- ✓ If the user removes all tasks and exits naturally, use else to print "All tasks completed. Goodbye!".
- ✓ Use pass for future enhancements (like saving tasks to a file).

## Mini Project 2: Simple Banking System (Using while, continue, break, pass, else)

### Task:

Create a Banking System that:

- Starts with a balance of ₹10,000.
- Allows the user to deposit and withdraw money.
- Displays the current balance after every transaction.
- Exits when the user types "quit".

### Conditions to Use:

- ✓ Use an infinite while loop to keep the banking system running.

- ✓ If the user enters a negative deposit/withdrawal amount, use continue to ask again.
- ✓ If the user tries to withdraw more than the balance, display "Insufficient funds" and continue.
- ✓ If the user types "quit", break the loop.
- ✓ If the loop exits naturally, print "Thank you for using our banking system!" using else.
- ✓ Use pass for any future enhancements (like adding a PIN system).

## Day 27

### Functions in Python

A function is a block of reusable code that performs a specific task. Instead of writing the same code multiple times, we can define a function once and call it whenever needed.

#### Why Use Functions?

- Code Reusability – Write once, use multiple times.
- Modularity – Break a big program into smaller, manageable parts.
- Readability – Makes the code cleaner and easier to understand.
- Avoid Redundancy – No need to repeat the same code.

#### Defining a Function

Functions in Python are defined using the def keyword.

**Syntax:**

```
def function_name(parameters):
    # Function body
    return value # Optional
```

- `function_name`: Name of the function.
- `parameters`: Input values (optional).
- `return`: Sends back a result (optional).

**Example: Simple Function**

```
def greet():
    print("Hello! Welcome to Python.")

greet() # Calling the function
```

**Output:**

Hello! Welcome to Python.

**def Keyword**

The `def` keyword is used to define a function in Python.

**Example: Function with Parameters**

```
def add(a, b):
    return a + b

result = add(5, 3)
print(result) # Output: 8
```

## Use of pass Statement in Function

The pass statement is used as a placeholder when defining a function but not implementing it yet.

### Example: Using pass

```
def future_function():
    pass # Placeholder, will add logic later
```

## Return Statement

The return statement sends back a value from a function.

### Example: Function with return

```
def square(n):
    return n * n

result = square(4)
print(result) # Output: 16
```

## Global and Local Variables

- Local Variable: Defined inside a function and accessible only within that function.
- Global Variable: Defined outside a function and accessible throughout the program.

### Example: Global vs Local Variable

```
x = 10 # Global variable
```

```
def my_function():
    y = 5 # Local variable
```

```

print(y)

my_function()
print(x) # Works
# print(y) # Error (y is local)

```

## Recursion in Python

Recursion is when a function calls itself to solve a problem.

### Example: Factorial Using Recursion

```

def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n - 1)

```

```
print(factorial(5)) # Output: 120
```

### \*args and \*\*kwargs in Functions

- \*args: Allows passing multiple arguments as a tuple.
- \*\*kwargs: Allows passing multiple keyword arguments as a dictionary.

### Example: Using \*args

```

def add_numbers(*args):
    return sum(args)

```

```
print(add_numbers(1, 2, 3, 4)) # Output: 10
```

### Example: Using \*\*kwargs

```

def student_details(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")
student_details(name="John", age=22, course="Python")

```

**Output:**

name: John  
age: 22  
course: Python

**self as Default Argument**

In object-oriented programming (OOP), self represents the instance of the class.

**✓ Example: self in a Class Function**

```
class Person:  
    def __init__(self, name):  
        self.name = name # Using self  
  
    def greet(self):  
        print(f"Hello, my name is {self.name}")  
  
p1 = Person("Alice")  
p1.greet()
```

**Output:**

Hello, my name is Alice

**First-Class Functions**

In Python, functions are treated as first-class citizens, meaning they can be assigned to variables, passed as arguments, or returned from other functions.

**Example: Function as Argument**

```
def shout(text):
    return text.upper()

def greet(func):
    print(func("hello"))

greet(shout) # Output: HELLO
```

**Lambda Function (Anonymous Function)**

A lambda function is a small anonymous function defined using the `lambda` keyword.

**Example: Lambda Function**

```
square = lambda x: x * x
print(square(5)) # Output: 25
```

**map(), reduce(), and filter() Functions**

These are built-in functions used for efficient data processing.

**1. map() Function**

Applies a function to all items in an iterable.

```
numbers = [1, 2, 3, 4]
squared = list(map(lambda x: x*x, numbers))
print(squared) # Output: [1, 4, 9, 16]
```

## 2. filter() Function

Filters elements based on a condition.

```
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers) # Output: [2, 4, 6]
```

## 3. reduce() Function

Reduces a list to a single value.

```
from functools import reduce
numbers = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, numbers)
print(product) # Output: 24
```

## Inner Function

A function defined inside another function.

### Example: Inner Function

```
def outer_function():
    def inner_function():
        print("I am an inner function.")
    inner_function()

outer_function()
```

### Output:

I am an inner function.

## Summary of Key Concepts

Concept	Explanation
def keyword	Defines a function
pass statement	Placeholder in a function
return	Returns a value from a function
Global & Local Variables	Global = accessible everywhere, Local = only inside function
Recursion	Function calls itself
*args	Pass multiple arguments as a tuple
**kwargs	Pass multiple keyword arguments as a dictionary
self	Represents the instance of the class
First-Class Functions	Functions can be assigned to variables, passed, or returned
Lambda Function	Anonymous function using lambda
map(), filter(), reduce()	Built-in functions for data processing
Inner Function	Function inside another function

## Mini Project 1: Student Management System

Topics Covered:

- ✓ def keyword
- ✓ pass statement
- ✓ return statement
- ✓ Global and Local Variables
- ✓ \*args and \*\*kwargs
- ✓ self as Default Argument
- ✓ First-Class Function

✓ Lambda Function

✓ map(), reduce(), filter()

✓ Inner Function

### **Project Description:**

Create a Student Management System that allows adding students, displaying student details, and calculating the average score.

### **Implementation:**

```
from functools import reduce
```

```
class Student:
```

```
    students_list = [] # Global Variable
```

```
    def __init__(self, name, age, *scores):
```

```
        self.name = name # Using self
```

```
        self.age = age
```

```
        self.scores = scores
```

```
        Student.students_list.append(self) # Store student data
```

```
    def get_average(self):
```

```
        return sum(self.scores) / len(self.scores)
```

```
    @staticmethod
```

```
    def filter_passing_students():
```

```
        return list(filter(lambda s: s.get_average() >= 50, Student.students_list))
```

```
    @staticmethod
```

```
    def get_highest_score():
```

```
        return reduce(lambda x, y: x if x.get_average() > y.get_average() else y,
```

```
Student.students_list)
```

```

@staticmethod
def display_students():
    for student in Student.students_list:
        print(f"Name: {student.name}, Age: {student.age}, Avg Score:
{student.get_average()}")


# Adding Student Records
s1 = Student("Alice", 20, 45, 78, 90)
s2 = Student("Bob", 21, 88, 92, 85)
s3 = Student("Charlie", 22, 40, 35, 30)

# Display Students
Student.display_students()

# Filter Passing Students
passing_students = Student.filter_passing_students()
print("\nPassing Students:")
for student in passing_students:
    print(student.name)

# Find Student with Highest Score
top_student = Student.get_highest_score()
print(f"\nTop Student: {top_student.name} with Avg Score:
{top_student.get_average()}")

```

## Mini Project 2: Recursive To-Do List Manager

### Topics Covered:

- ✓ def keyword
- ✓ pass statement
- ✓ return statement

- ✓ Recursion in Python
- ✓ \*args and \*\*kwargs
- ✓ First-Class Function
- ✓ Lambda Function
- ✓ map(), filter()
- ✓ Inner Function

### **Project Description:**

A To-Do List Manager where users can add, remove, and display tasks using recursion.

### **Implementation:**

```
class ToDoList:
    def __init__(self):
        self.tasks = []

    def add_task(self, task):
        self.tasks.append(task)

    def remove_task(self, task):
        self.tasks = list(filter(lambda t: t != task, self.tasks))

    def show_tasks(self):
        if not self.tasks:
            print("No tasks available.")
        else:
            print("\nYour Tasks:")
            list(map(lambda t: print(f"- {t}"), self.tasks))

    def manage_list(self):
        def menu():
```

```

print("\n1. Add Task\n2. Remove Task\n3. Show Tasks\n4. Exit")
return input("Choose an option: ")

def process_choice(choice):
    if choice == '1':
        task = input("Enter task: ")
        self.add_task(task)
    elif choice == '2':
        task = input("Enter task to remove: ")
        self.remove_task(task)
    elif choice == '3':
        self.show_tasks()
    elif choice == '4':
        print("Exiting To-Do List...")
        return
    else:
        print("Invalid choice, try again!")

    self.manage_list() # Recursion to continue until exit

process_choice(menu()) # Start Recursion

```

```

# Run To-Do List Manager
todo = ToDoList()
todo.manage_list()

```

### Features of Both Projects

- ✓ Uses **functions** for modular code
- ✓ Demonstrates **recursion** (To-Do List menu)
- ✓ Implements **lambda functions** and map(), filter(), reduce()

- ✓ Utilizes **global & local variables**
- ✓ Uses **OOP concepts** with self

## Tasks

1. Define a function `greet_user()` that prints "Hello, User!" when called.
2. Create a function `calculate_sum(a, b)` that takes two numbers as arguments and returns their sum.
3. Write a function `check_positive(num)` that takes a number as input. If the number is positive, return "Positive", else use the `pass` statement.
4. Create a function `find_max(a, b, c)` that takes three numbers and returns the maximum among them using the `return` statement.
5. Demonstrate global and local variables by defining a global variable `count = 10` and modifying it inside a function using `global count`.
6. Write a function `modify_variable()` where a local variable `message = "Local Scope"` is declared. Print this inside and outside the function to show scope difference.
7. Write a recursive function `factorial(n)` to find the factorial of a number.
8. Create a recursive function `fibonacci(n)` that returns the nth Fibonacci number.
9. Write a function `sum_numbers(*args)` that takes multiple numbers and returns their sum.
10. Create a function `print_student_details(**kwargs)` that takes keyword arguments like `name, age, and grade`, and prints them.
11. Create a function `apply_operation(func, a, b)` that takes a function and two numbers as arguments. Pass `lambda x, y: x + y` as an argument and return the sum.

12. Write a function outer\_function() that defines an inner function inner() inside it and returns "Hello from Inner Function".

13. Use map() function to double each element of a given list [1, 2, 3, 4, 5].

**Bonus Challenge:**

Convert all even numbers from a list [10, 15, 20, 25, 30] into strings using map() and lambda.

## **Mini Project 1: Student Grade Management System**

**Concepts Used:** def keyword, return statement, \*args, \*\*kwargs, recursion, lambda, map(), filter(), reduce().

**Task Description:**

- Create a **function** calculate\_average(\*args) that takes multiple subject marks and returns the average score.
- Use a lambda function inside map() to convert marks into grades ( $\geq 90$ : "A", 80-89: "B", 70-79: "C",  $< 70$ : "F").
- Create a filter function to find students who passed (grade is not "F").
- Use reduce() to find the highest score among all students.
- Implement recursion to print the first n students' details.
- Use \*\*kwargs to store student details dynamically (name, age, grade).

**Expected Output Example:**

Student: John, Age: 16, Average Score: 85, Grade: B

Passed Students: ['John', 'Emily']

Highest Score: 95

## Mini Project 2: Banking System with OOP and Functions

**Concepts Used:** self (OOP), global/local variables, inner functions, lambda, first-class functions.

### Task Description:

- Create a BankAccount class with attributes (name, balance).
- Define methods like deposit(amount), withdraw(amount), and get\_balance().
- Use an inner function inside withdraw() to check if withdrawal is possible.
- Use a lambda function for transaction fees (lambda amount: amount \* 0.02).
- Demonstrate global vs local variables by maintaining a global total\_transactions counter.
- Implement a first-class function to apply an interest rate function to a balance.

### Expected Output Example:

Account Holder: Alice

Deposited: 500

Withdrawn: 200

Balance: 300

Transaction Fee: 4

# Day 28

## Python Strings

### What is a String?

- A string in Python is a sequence of characters enclosed in single ('), double ("") or triple ('''' ''''') quotes.
- Strings are immutable, meaning their content cannot be changed once created.
- Python treats anything enclosed in quotes as a string.

### Python Strings - Detailed Explanation

Python provides powerful features to work with **strings**, including creation, access, modification, and formatting. Let's explore each topic step by step.

#### 1. Creating a String

A string is a sequence of characters enclosed in single ('), double ("") or triple ('''' ''''') quotes.

#### Example:

```
# Using single, double, and triple quotes
string1 = 'Hello'
string2 = "Python"
string3 = """Multiline
String""
```

```
print(string1) # Output: Hello
print(string2) # Output: Python
print(string3) # Output: Multiline String
```

Triple quotes (''' or ''''') allow multiline strings.

## 2. Accessing Characters in a String

You can access characters in a string using indexing. Python indexing starts from 0.

### Example:

```
text = "Python"
```

```
print(text[0]) # Output: P (First character)
print(text[3]) # Output: h (Fourth character)
print(text[-1]) # Output: n (Last character)
```

Negative indexing allows access from the end of the string (-1 is the last character).

## 3. String Immutability

Strings in Python are immutable, meaning they cannot be changed after creation.

### Example:

```
text = "Hello"
```

```
# Trying to change a character (This will cause an error)
text[0] = "M" # ✗ TypeError: 'str' object does not support item assignment
```

Since strings are immutable, to modify a string, we must create a new string.

### Correct Approach (Reassigning a New String):

```
text = "Hello"
new_text = "M" + text[1:] # Changing 'H' to 'M'
print(new_text) # Output: Mello
```

#### 4. Deleting a String

We cannot delete individual characters, but we can delete the entire string using the `del` keyword.

##### Example:

```
text = "Python"
del text # Deletes the entire string

print(text) # ✗ NameError: name 'text' is not defined
```

#### 5. Updating a String

Since strings are immutable, you cannot modify them directly. Instead, you can reassign a new string.

##### Example:

```
text = "Hello"
text = text + " World!" # Concatenation (Creating a new string)

print(text) # Output: Hello World!
```

#### 6. Common String Methods

Python provides several built-in string methods for string manipulation.

Method	Description	Example
<code>len()</code>	Returns string length	<code>len("Hello") → 5</code>
<code>lower()</code>	Converts to lowercase	<code>"PYTHON".lower() → "python"</code>
<code>upper()</code>	Converts to uppercase	<code>"hello".upper() → "HELLO"</code>
<code>strip()</code>	Removes spaces from start & end	<code>" Hello ".strip() → "Hello"</code>
<code>replace()</code>	Replaces substring	<code>"apple".replace("a", "o") → "opple"</code>
<code>split()</code>	Splits string into a list	<code>"a,b,c".split(",") → ['a', 'b', 'c']</code>

join()	Joins elements of a list	"-".join(['a', 'b']) → "a-b"
find()	Finds index of a substring	"hello".find("l") → 2
count()	Counts occurrences of a substring	"banana".count("a") → 3

**Example of Using String Methods:**

```
text = " Hello Python! "
```

```
print(text.strip()) # Removes spaces → "Hello Python!"
```

```
print(text.upper()) # Converts to uppercase → " HELLO PYTHON! "
```

```
print(text.replace("Python", "World")) # Replaces text → " Hello World! "
```

**7. Concatenating and Repeating Strings**

**Concatenation (+):** Combines two strings.

**Repetition (\*):** Repeats a string multiple times.

**Example:**

```
text1 = "Hello"
```

```
text2 = "Python"
```

```
# Concatenation
```

```
result = text1 + " " + text2
```

```
print(result) # Output: Hello Python
```

```
# Repeating a string
```

```
print("Hi! " * 3) # Output: Hi! Hi! Hi!
```

## 8. Formatting Strings

String formatting allows inserting values inside a string dynamically.

### Using f-strings (Recommended):

```
name = "Alice"
```

```
age = 25
```

```
print(f"My name is {name} and I am {age} years old.")
```

# Output: My name is Alice and I am 25 years old.

### Using .format() Method:

```
print("My name is {} and I am {} years old.".format(name, age))
```

# Output: My name is Alice and I am 25 years old.

### Using % Operator (Old Method):

```
print("My name is %s and I am %d years old." % (name, age))
```

# Output: My name is Alice and I am 25 years old.

## Summary

- ✓ Creating Strings - Strings are enclosed in ', ", or """ """.
- ✓ Accessing Characters - Use indexing (text[0], text[-1]).
- ✓ Immutability - Strings cannot be changed, only reassigned.
- ✓ Deleting Strings - del deletes a string variable.
- ✓ Updating Strings - Create a new string instead of modifying.
- ✓ Common Methods - len(), upper(), lower(), strip(), replace(), etc.
- ✓ Concatenation & Repetition - + joins, \* repeats.
- ✓ String Formatting - Use f-strings, .format(), or % formatting.

## Mini Project 1: Student Profile Card Generator

### Concepts Used:

- ✓ Creating a String
- ✓ Accessing Characters in a String
- ✓ String Immutability
- ✓ Updating a String
- ✓ Common String Methods
- ✓ Formatting Strings

### Project Description:

Create a Python program that takes a student's name, age, course, and university name as input and generates a formatted profile card.

### Expected Output Example:

🎓 Student Profile 🎓

---

Name : Alice Johnson  
Age : 21  
Course : Computer Science  
University: Harvard University

---

Note: Keep working hard and learning new things! 💡

### Code Implementation:

```
# Taking input from the user
name = input("Enter Student Name: ").strip().title()
age = input("Enter Age: ").strip()
course = input("Enter Course Name: ").strip().title()
```

```
university = input("Enter University Name: ").strip().title()
```

```
# Generating formatted profile
```

```
profile = f"""\n
```

```
    🎓 Student Profile 🎓\n-----
```

```
    Name : {name}
```

```
    Age : {age}
```

```
    Course : {course}
```

```
    University: {university}\n-----
```

```
    Note: Keep working hard and learning new things! 💪
```

```
    """
```

```
# Printing the formatted student profile
```

```
print(profile)
```

## Mini Project 2: Fun String Manipulator Tool

Concepts Used:

- ✓ Creating a String
- ✓ Accessing Characters in a String
- ✓ String Immutability
- ✓ Deleting a String
- ✓ Updating a String
- ✓ Common String Methods
- ✓ Concatenating & Repeating Strings
- ✓ Formatting Strings

**Project Description:**

Create a Python program that takes a sentence from the user and performs various string manipulations, displaying the results in a structured format.

**Expected Output Example:**

<sup>AB</sup><sub>CD</sub> Fun String Manipulator <sup>AB</sup><sub>CD</sub>

---

Original Sentence : Python is fun!

Uppercase : PYTHON IS FUN!

Lowercase : python is fun!

First Character : P

Last Character : !

Reversed Sentence : !nuf si nohtyP

Repeated Sentence : Python is fun!Python is fun!Python is fun!

Formatted Output : Learning Python is really FUN!

---

**Code Implementation:**

```
# Taking user input
sentence = input("Enter a sentence: ").strip()

# Performing string manipulations
uppercase = sentence.upper()
lowercase = sentence.lower()
first_char = sentence[0]
last_char = sentence[-1]
reversed_sentence = sentence[::-1]
repeated_sentence = sentence * 3

# Formatting a new string
formatted_output = f"Learning {sentence.split()[0]} is really FUN!"
```

```

# Displaying results
result = f"""
AB CD Fun String Manipulator AB CD
-----
Original Sentence : {sentence}
Uppercase      : {uppercase}
Lowercase      : {lowercase}
First Character : {first_char}
Last Character  : {last_char}
Reversed Sentence : {reversed_sentence}
Repeated Sentence : {repeated_sentence}
Formatted Output : {formatted_output}
-----
"""
print(result)

```

### **Summary:**

- Mini Project 1: Generates a student profile card using string manipulation techniques.
- Mini Project 2: Creates a fun string manipulator tool that applies multiple string operations.

### **Day 28 Tasks**

#### **Task 1: Create and Print a String**

Create a string variable called greeting and store "Hello, Python!" in it. Print the string.

**Expected Output:**

Hello, Python!

**Task 2:** Access Specific Characters in a String

Given the string "PythonProgramming", access and print:

The first character

The last character

The middle character

**Task 3:** Slice a String

Given the string "Python Developer", extract and print:

The word "Python"

The word "Developer"

The string in reverse order

**Task 4:** Try Modifying a String (String Immutability)

Given the string "Immutable", try to change the first letter to "A". Observe and explain the error message.

**Task 5:** Delete a String

Create a string variable and assign "Temporary String" to it. Then delete the variable and try printing it. What happens?

**Task 6:** Update a String

Given "Hello, World!", update it to "Hello, Python!" by slicing and concatenation.

**Expected Output:**

Hello, Python!

### **Task 7: Use String Methods**

Given the string "Python is Amazing!", apply and print the results of the following methods:

.upper()  
.lower()  
.title()  
.replace("Amazing", "Powerful")

### **Task 8: Check String Properties**

Given the string "Hello123", check and print whether it:

Contains only alphabets  
Contains only digits  
Contains both letters and numbers

### **Task 9: Concatenating and Repeating Strings**

Given "Python" and "Programming", concatenate them with a space in between and print the result.

Repeat the string "Python! " 5 times and print the result.

### **Task 10: Format a String Using f-strings**

Take user input for name and age, then print:

Hello, my name is <name> and I am <age> years old.

#### **Example Input:**

Enter your name: Alice

Enter your age: 25

**Expected Output:**

Hello, my name is Alice and I am 25 years old.

**Task 11:** Find and Replace a Word in a String

Given the sentence "I love Java!", replace "Java" with "Python" and print the updated sentence.

**Expected Output:**

I love Python!

**Task 12:** Count the Occurrences of a Character

Given the string "banana", count how many times the letter "a" appears.

**Expected Output:**

The letter 'a' appears 3 times.

**Task 13:** Reverse Words in a Sentence

Given the sentence "Python is fun", reverse the order of words and print:

**Expected Output:**

fun is Python

**Bonus Challenge Task:**

Take a user's full name as input and print it in the format:

First Name: <First Part>

Last Name: <Last Part>

Reversed Name: <Last Part> <First Part>

**Example:**

Enter your full name: John Doe

**Output:**

First Name: John

Last Name: Doe

Reversed Name: Doe John

**Summary:**

These 13 tasks will help strengthen your string manipulation skills in Python. Try solving them step by step and test different string methods!

**Mini Project 1: User Profile Formatter****Description:**

Create a Python program that asks the user for their full name, age, and favorite quote. Then, format and display the information in a structured manner using string methods, concatenation, and f-strings.

**Steps to Implement:**

1. Take user input for:
  - a. Full Name
  - b. Age
  - c. Favorite Quote
2. Capitalize the first letter of each word in the name (using `.title()`).
3. Ensure that the age is in string format (use `str()` if needed).
4. Convert the favorite quote to **uppercase** using `.upper()`.
5. Display the formatted output like this:

**❖ User Profile:**

-----  
Name : John Doe

Age : 25

Favorite Quote : "SUCCESS COMES TO THOSE WHO WORK FOR IT."

### **Expected Output Example:**

Enter your full name: alice johnson

Enter your age: 22

Enter your favorite quote: keep moving forward

### **❖ User Profile:**

Name : Alice Johnson

Age : 22

Favorite Quote : "KEEP MOVING FORWARD."

## **Mini Project 2: Simple Password Generator**

### **Description:**

Create a simple password generator using string manipulation techniques. The password will be created based on a user's name and a keyword they provide.

### **Steps to Implement:**

1. Take user input for:
  - a. First Name
  - b. Last Name
  - c. A secret keyword
2. Use string slicing to take:
  - a. First three letters of the first name
  - b. Last three letters of the last name
  - c. Reverse the secret keyword
3. Concatenate these values and convert them to a mix of uppercase and lowercase letters.

4. Display the generated password in a formatted way.

**Expected Output Example:**

Enter your first name: David

Enter your last name: Johnson

Enter a secret keyword: python

 Your generated password is: Davsonnohtyp

(Explanation: "Dav" from David, "son" from Johnson, and "python" reversed → "nohtyp")

**Learning Outcome:**

- ✓ String Manipulation (creating, updating, deleting, accessing)
- ✓ String Methods (.upper(), .lower(), .title(), .replace())
- ✓ String Concatenation & Formatting
- ✓ String Slicing & Immutability

These projects are simple yet practical!

# Day 29

## Python Lists – Explanation with Examples

### What is a List in Python?

A list in Python is an ordered, mutable collection of items that can hold different data types (integers, strings, floats, other lists, etc.).

Lists are defined using square brackets [] and elements are separated by commas.

#### Example:

```
my_list = [10, "hello", 3.14, True]  
print(my_list) # Output: [10, 'hello', 3.14, True]
```

### 1. Creating a List in Python

You can create a list by **enclosing elements in square brackets []**.

#### Example:

```
# Empty list  
empty_list = []
```

```
# List with different data types  
numbers = [1, 2, 3, 4, 5]  
fruits = ["apple", "banana", "cherry"]  
mixed_list = [10, "hello", 3.5, True]
```

```
print(fruits) # Output: ['apple', 'banana', 'cherry']
```

## 2. Accessing List Elements

You can access list elements using indexing (0-based indexing) and negative indexing.

### Example:

```
fruits = ["apple", "banana", "cherry"]
```

```
# Accessing elements using positive indexing
```

```
print(fruits[0]) # Output: apple
```

```
print(fruits[1]) # Output: banana
```

```
# Accessing elements using negative indexing
```

```
print(fruits[-1]) # Output: cherry
```

```
print(fruits[-2]) # Output: banana
```

## 3. Adding Elements into a List

You can add elements to a list using:

- `append()` → Adds an item at the end.
- `insert(index, element)` → Adds an item at a specific index.
- `extend(iterable)` → Adds multiple items.

### Example:

```
fruits = ["apple", "banana"]
```

```
# Append
```

```
fruits.append("cherry")
```

```
print(fruits) # Output: ['apple', 'banana', 'cherry']
```

```
# Insert at index 1
```

```
fruits.insert(1, "mango")
```

```

print(fruits) # Output: ['apple', 'mango', 'banana', 'cherry']

# Extend with multiple items
fruits.extend(["grape", "orange"])
print(fruits) # Output: ['apple', 'mango', 'banana', 'cherry', 'grape', 'orange']

```

## 4. Updating Elements in a List

Lists are **mutable**, meaning you can modify elements after creation.

**Example:**

```

fruits = ["apple", "banana", "cherry"]
fruits[1] = "mango"
print(fruits) # Output: ['apple', 'mango', 'cherry']

```

## 5. Removing Elements from a List

You can remove elements using:

- `remove(value)` → Removes the first occurrence of a value.
- `pop(index)` → Removes an item at a specific index.
- `del list[index]` → Deletes an element or entire list.
- `clear()` → Empties the list.

**Example:**

```

fruits = ["apple", "banana", "cherry", "banana"]

# Remove specific value
fruits.remove("banana")
print(fruits) # Output: ['apple', 'cherry', 'banana']

```

```
# Remove using pop (default: last element)
fruits.pop()
print(fruits) # Output: ['apple', 'cherry']

# Remove using index
del fruits[0]
print(fruits) # Output: ['cherry']

# Clear the list
fruits.clear()
print(fruits) # Output: []
```

## 6. Iterating Over Lists

You can loop through a list using a for loop.

### Example:

```
fruits = ["apple", "banana", "cherry"]
```

```
for fruit in fruits:
    print(fruit)
```

### Output:

```
apple
banana
cherry
```

## 7. Nested Lists in Python

A nested list is a list inside another list.

**Example:**

```
nested_list = [[1, 2, 3], ["apple", "banana"], [True, False]]
# Accessing an element inside a nested list
print(nested_list[1][0]) # Output: apple
print(nested_list[2][1]) # Output: False
```

**8. Python List Operations & Programs****List Concatenation**

You can **join two lists** using **+**.

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
```

```
new_list = list1 + list2
print(new_list) # Output: [1, 2, 3, 4, 5, 6]
```

**List Repetition**

You can **repeat a list multiple times** using **\***.

```
numbers = [1, 2, 3]
print(numbers * 2) # Output: [1, 2, 3, 1, 2, 3]
```

**List Slicing**

Extract a **portion of a list**.

```
fruits = ["apple", "banana", "cherry", "grape"]
print(fruits[1:3]) # Output: ['banana', 'cherry']
print(fruits[:2]) # Output: ['apple', 'banana']
```

## Checking if an Item Exists

Use `in` to check if an item is in a list.

```
fruits = ["apple", "banana", "cherry"]
print("banana" in fruits) # Output: True
print("grape" in fruits) # Output: False
```

## Summary of List Operations

Operation	Method	Example
Creating a list	<code>list = [1, 2, 3]</code>	<code>[1, 2, 3]</code>
Access elements	<code>list[index]</code>	<code>list[0] → 1</code>
Add elements	<code>append(), insert(), extend()</code>	<code>list.append(4)</code>
Update elements	<code>list[index] = new_value</code>	<code>list[1] = 10</code>
Remove elements	<code>remove(), pop(), del, clear()</code>	<code>list.pop(1)</code>
Loop through list	<code>for item in list:</code>	Iteration
Nested lists	<code>list = [[1, 2], [3, 4]]</code>	Access like <code>list[0][1]</code>
List operations	Concatenation, slicing, checking existence	<code>list1 + list2</code>

## Conclusion

Python lists are versatile and powerful, allowing you to store, modify, and manipulate data easily.

Mastering lists is essential for data handling in Python programs.

## Mini Project 1: Student Grade Tracker

**Concepts Used:** Creating a list, accessing elements, adding/updating/removing elements, iterating over lists.

### Project Description:

Create a Python program that allows a teacher to store student names and their grades. The program should allow:

- Adding new students and grades
- Updating a student's grade
- Removing a student from the list
- Displaying all students and their grades

### Sample Code:

```
# Initialize an empty list to store student data
students = []

# Function to add a student
def add_student(name, grade):
    students.append([name, grade])
    print(f"Student {name} with grade {grade} added!")

# Function to update a grade
def update_grade(name, new_grade):
    for student in students:
        if student[0] == name:
            student[1] = new_grade
            print(f"Grade for {name} updated to {new_grade}")
            return
    print("Student not found!")
```

```
# Function to remove a student
def remove_student(name):
    global students
    students = [student for student in students if student[0] != name]
    print(f"Student {name} removed!")

# Function to display all students
def display_students():
    print("\nStudent List:")
    for student in students:
        print(f"{student[0]}: {student[1]}")

# Menu-driven program
while True:
    print("\n1. Add Student\n2. Update Grade\n3. Remove Student\n4. Display
Students\n5. Exit")
    choice = int(input("Enter your choice: "))

    if choice == 1:
        name = input("Enter student name: ")
        grade = input("Enter grade: ")
        add_student(name, grade)
    elif choice == 2:
        name = input("Enter student name: ")
        new_grade = input("Enter new grade: ")
        update_grade(name, new_grade)
    elif choice == 3:
        name = input("Enter student name to remove: ")
        remove_student(name)
    elif choice == 4:
        display_students()
```

```
elif choice == 5:  
    print("Exiting program...")  
    break  
else:  
    print("Invalid choice! Please try again.")
```

### Features:

- ✓ Add students and their grades
- ✓ Update a student's grade
- ✓ Remove students from the list
- ✓ Display all students with grades

## Mini Project 2: To-Do List Manager ✓

**Concepts Used:** Creating a list, adding/updating/removing elements, iterating over lists, list operations.

### Project Description:

Create a simple To-Do List Manager where users can:

- Add tasks to their to-do list
- Mark tasks as completed
- Remove tasks from the list
- View all pending tasks

### Sample Code:

```
# Initialize an empty to-do list  
todo_list = []
```

```
# Function to add a task
def add_task(task):
    todo_list.append(task)
    print(f"Task '{task}' added!")

# Function to remove a task
def remove_task(task):
    if task in todo_list:
        todo_list.remove(task)
        print(f"Task '{task}' removed!")
    else:
        print("Task not found!")

# Function to display tasks
def display_tasks():
    if not todo_list:
        print("\nNo tasks to show!")
    else:
        print("\nTo-Do List:")
        for index, task in enumerate(todo_list, start=1):
            print(f"{index}. {task}")

# Menu-driven program
while True:
    print("\n1. Add Task\n2. Remove Task\n3. View Tasks\n4. Exit")
    choice = int(input("Enter your choice: "))

    if choice == 1:
        task = input("Enter task: ")
        add_task(task)
    elif choice == 2:
```

```
task = input("Enter task to remove: ")
remove_task(task)
elif choice == 3:
    display_tasks()
elif choice == 4:
    print("Exiting program...")
    break
else:
    print("Invalid choice! Please try again.")
```

**Features:**

- ✓ Add tasks to the list
- ✓ Remove tasks from the list
- ✓ View all pending tasks

**Day 29 tasks**

1. Create a list of 5 favorite movies and print it.
2. Access the 2nd and 4th elements from a given list and print them.
3. Add three new items to an existing list of fruits using .append() and .insert().
4. Update the value of an element in a list (e.g., change a book name in a book list).
5. Remove a specific item from a list using .remove() and del.
6. Iterate through a list of numbers and print only the even numbers.
7. Iterate through a list of names and print them in uppercase.
8. Reverse a given list using slicing ([::-1]) and print the reversed list.

9. Create a nested list representing students and their marks, then print the marks of a specific student.
10. Flatten a nested list (convert [[1, 2], [3, 4], [5, 6]] into [1, 2, 3, 4, 5, 6]).
11. Sort a list of numbers in ascending and descending order using .sort().
12. Find the maximum and minimum values in a list of numbers.
13. Write a program to remove duplicates from a given list without using set().

## **Mini Project 1: Student Management System (Using Lists)**

**Concepts Used:** Creating, Accessing, Updating, Removing, and Iterating Over Lists.

### **Task:**

Create a Student Management System where:

- ✓ A list stores student names.
- ✓ Allow users to add a new student.
- ✓ Allow users to remove a student.
- ✓ Allow users to update a student's name.
- ✓ Display all student names using a loop.
- ✓ Ensure the system runs until the user decides to exit.

### **Expected Output Example:**

1. Add Student
2. Remove Student
3. Update Student Name
4. Show All Students
5. Exit

Enter your choice: 1

Enter Student Name: John

Student added successfully!

Enter your choice: 4

Student List: ['John']

## Mini Project 2: Shopping Cart System

**Concepts Used:** Lists, Nested Lists, Iteration, Adding & Removing Elements.

### Task:

Create a Shopping Cart System where:

- ✓ The cart is a list containing nested lists ([product\_name, price]).
- ✓ Users can add products to the cart with their price.
- ✓ Users can remove a product by name.
- ✓ Display the total number of items and the total price.
- ✓ Allow users to view all items in the cart.

### Expected Output Example:

1. Add Product
2. Remove Product
3. View Cart
4. Checkout

Enter your choice: 1

Enter Product Name: Laptop

Enter Price: 50000

Product added successfully!

Enter your choice: 3

Shopping Cart: [['Laptop', 50000]]

Total Price: 50000

## Day 30

### What is a Tuple in Python?

A tuple is an ordered, immutable collection in Python that can hold multiple values. Tuples are similar to lists, but they cannot be changed (immutable) after creation. Tuples allow storing different data types and are faster than lists.

#### Syntax of a Tuple:

```
my_tuple = (1, 2, 3, "Hello", 4.5)
```

#### 1. Creating a Tuple

Tuples are created using parentheses () and can store elements of different data types.

##### Example:

```
empty_tuple = () # Empty tuple  
single_element_tuple = (5,) # Tuple with one element (comma is needed)  
multi_tuple = (10, "Python", 3.14) # Tuple with multiple values
```

**Note:** Even though tuples use parentheses, they can be created without them.

```
my_tuple = 1, 2, 3 # Tuple without parentheses
```

## 2. Python Tuple Operations

Tuples support indexing, slicing, and concatenation just like lists.

### Basic Operations:

```
t1 = (1, 2, 3)
```

```
t2 = (4, 5, 6)
```

```
print(len(t1)) # Length of tuple: 3  
print(t1 + t2) # Concatenation: (1, 2, 3, 4, 5, 6)  
print(t1 * 2) # Repetition: (1, 2, 3, 1, 2, 3)
```

## 3. Accessing Elements in Tuples

Tuples use zero-based indexing to access elements.

### Example:

```
t = ("apple", "banana", "cherry")  
print(t[0]) # Output: apple  
print(t[-1]) # Output: cherry (negative indexing)
```

## 4. Concatenation of Tuples

Tuples can be joined using the + operator.

### Example:

```
t1 = (1, 2, 3)  
t2 = (4, 5, 6)  
t3 = t1 + t2  
print(t3) # Output: (1, 2, 3, 4, 5, 6)
```

## 5. Slicing of Tuples

Tuples support slicing using the format:

```
tuple[start:end:step]
```

### Example:

```
t = (10, 20, 30, 40, 50, 60)
print(t[1:4]) # Output: (20, 30, 40)
print(t[:3]) # Output: (10, 20, 30)
print(t[::-2]) # Output: (10, 30, 50)
```

## 6. Deleting a Tuple

Since tuples are immutable, you cannot delete individual elements, but you can delete the entire tuple using del.

### Example:

```
t = (1, 2, 3)
del t
# print(t) # This will give an error because t is deleted
```

## 7. Tuple Built-In Methods

Tuples have limited methods since they are immutable.

### Common Tuple Methods:

```
t = (1, 2, 3, 4, 1, 2, 1)
print(t.count(1)) # Output: 3 (Counts occurrences of 1)
print(t.index(3)) # Output: 2 (Finds index of 3)
```

## 8. Tuple Built-In Functions

Some useful built-in functions that work on tuples:

### Examples:

```
t = (10, 5, 20, 8)
print(len(t)) # Output: 4
print(max(t)) # Output: 20
print(min(t)) # Output: 5
print(sum(t)) # Output: 43
```

## 9. Tuples vs Lists

Feature	Tuple (tuple)	List (list)
<b>Mutability</b>	Immutable	Mutable
<b>Speed</b>	Faster	Slower
<b>Memory Use</b>	Less memory	More memory
<b>Methods</b>	Limited	More methods

### Example Comparison:

```
# List (Mutable)
my_list = [1, 2, 3]
my_list[0] = 100
print(my_list) # Output: [100, 2, 3]

# Tuple (Immutable)
my_tuple = (1, 2, 3)
# my_tuple[0] = 100 # ✗ This will cause an error
```

## **Summary:**

- Tuple is an immutable collection of items.
- Elements can be accessed using indexing & slicing.
- Tuples support operations like concatenation & repetition.
- Methods like count() and index() are available.
- Tuples use less memory and are faster than lists.

## **Mini Project 1: Student Grades Management using Tuples**

**Concepts Covered:** Creating a Tuple, Accessing Tuples, Tuple Operations, Tuple Built-in Methods & Functions.

### **Project Description:**

Create a Python program that stores students' names and their grades in tuples. The program should allow users to:

1. View all student grades.
2. Get the highest, lowest, and average grade using tuple functions.
3. Access a student's grade using their index.

### **Example Implementation:**

```
# Tuple storing student names and grades
students = ("Alice", "Bob", "Charlie", "David")
grades = (85, 92, 78, 90)
```

```
# Display all students and grades
print("Student Grades:")
for i in range(len(students)):
    print(f"{students[i]}: {grades[i]}")
```

```
# Finding highest, lowest, and average grade
print("\nStatistics:")
print(f"Highest Grade: {max(grades)}")
print(f"Lowest Grade: {min(grades)}")
print(f"Average Grade: {sum(grades) / len(grades):.2f}")

# Accessing a student's grade
index = students.index("Charlie")
print(f"\nCharlie's Grade: {grades[index]}")
```

## **Mini Project 2: Inventory Management System using Tuples**

**Concepts Covered:** Tuple Creation, Accessing Elements, Concatenation, Slicing, and Tuple Methods.

### **Project Description:**

Create a simple inventory management system using tuples where:

1. Items and their prices are stored in tuples.
2. Users can view all items and prices.
3. Search for a specific item.
4. Remove an item from inventory (by reassigning a new tuple).

### **Example Implementation:**

```
# Inventory Tuple (Item Name, Price)
inventory = (("Laptop", 70000), ("Mouse", 1500), ("Keyboard", 2500), ("Monitor", 12000))

# Display all inventory items
print("Inventory Items:")
for item in inventory:
```

```
print(f"{item[0]} - Rs.{item[1]})

# Searching for an item
search_item = "Mouse"
found = next((item for item in inventory if item[0] == search_item), None)

if found:
    print(f"\n{search_item} is available at Rs.{found[1]}")
else:
    print(f"\n{search_item} is not available in the inventory.")

# Removing an item from inventory
remove_item = "Keyboard"
inventory = tuple(item for item in inventory if item[0] != remove_item)
print("\nUpdated Inventory (After Removing Keyboard):")
print(inventory)
```

## Day 30 Tasks

### 1. Create a Tuple:

Create a tuple with five different fruits and print it.

### 2. Tuple Operations - Indexing & Length:

Access the third element of the tuple and print its length using len().

### 3. Accessing Tuple Elements:

Create a tuple with 5 numbers and access the first and last elements using indexing.

### 4. Concatenation of Tuples:

Create two tuples, one with numbers and another with names. Concatenate them into a single tuple.

**5. Tuple Slicing:**

Create a tuple of numbers from 1 to 10 and extract a slice containing elements from index 2 to 7.

**6. Modifying a Tuple (Indirectly):**

Convert a tuple into a list, modify an element, and convert it back to a tuple.

**7. Deleting a Tuple:**

Create a tuple, delete it using del, and try to print it to observe the error.

**8. Using Tuple Methods:**

Create a tuple with repeated elements and use .count() to find how many times an element appears.

**9. Tuple Built-In Functions:**

Create a tuple of numbers and find the maximum, minimum, and sum of elements using max(), min(), and sum().

**10. Tuple vs List - Mutability Test:**

Create a tuple and a list with the same elements, try modifying both, and explain the result.

**11. Check if an Element Exists in a Tuple:**

Write a program that checks whether a given number exists in a tuple or not.

**12. Unpacking a Tuple:**

Create a tuple of four colors and unpack them into four different variables. Print each variable.

**13. Iterate Over a Tuple:**

Write a Python program to iterate through a tuple containing names and print each name in uppercase.

## Mini Projects

### 1. Student Marks Analyzer using Tuples

**Concept Covered:** Creating a Tuple, Accessing Elements, Tuple Operations, Tuple Functions (max(), min(), sum()), Tuple vs List

**Task:**

- Create a tuple containing marks of 5 subjects for a student.
- Calculate and print the total marks, highest marks, lowest marks, and average marks using tuple functions.
- Convert the tuple to a list, modify one of the subject marks, and convert it back to a tuple.

**Bonus:** Ask the user to input marks dynamically using `input()`, store them in a tuple, and perform the same analysis.

### Shopping Cart System using Tuples

**Concept Covered:** Tuple Creation, Tuple Concatenation, Tuple Iteration, Tuple Slicing, Tuple Methods

**Task:**

- Create a tuple containing different product names in a shopping cart.
- Allow the user to view all products in the cart.
- Allow the user to add a new product (Convert tuple to list, add an item, convert it back to a tuple).
- Allow the user to remove a product (Convert tuple to list, remove an item, convert it back to a tuple).
- Find how many times a specific product appears in the cart using `.count()`.
- Display only the first three items using tuple slicing.

**Bonus:** Display a confirmation message when a product is added or removed.

# Day 31

## What is a Dictionary in Python?

A dictionary in Python is an unordered collection of key-value pairs. It allows fast lookups, insertions, and deletions because it is implemented using hash tables. Dictionaries are mutable, meaning you can update, add, or remove elements dynamically.

### Syntax of a Dictionary:

```
my_dict = {"name": "Alice", "age": 25, "city": "New York"}
```

### Example:

```
student = {
    "name": "John",
    "age": 20,
    "course": "Computer Science"
}
print(student) # Output: {'name': 'John', 'age': 20, 'course': 'Computer Science'}
```

## 1. Accessing Dictionary Items

We can access values in a dictionary using keys.

### Using Square Brackets ([]):

```
student = {"name": "John", "age": 20}
print(student["name"]) # Output: John
```

If the key does not exist, this method raises a `KeyError`.

**Using .get() Method (Safer Way):**

```
print(student.get("age")) # Output: 20
print(student.get("grade", "Not Available")) # Output: Not Available
```

If the key does not exist, .get() returns None or a default value.

**2. Adding and Updating Dictionary Items**

We can add new key-value pairs or update existing values using assignment (=).

**Example:**

```
student = {"name": "John", "age": 20}
```

```
# Adding a new key-value pair
student["course"] = "Computer Science"
```

```
# Updating an existing key
student["age"] = 21
```

```
print(student)
# Output: {'name': 'John', 'age': 21, 'course': 'Computer Science'}
```

**Using .update() Method (Multiple Updates at Once):**

```
student.update({"age": 22, "grade": "A"})
print(student)
# Output: {'name': 'John', 'age': 22, 'course': 'Computer Science', 'grade': 'A'}
```

### 3. Removing Dictionary Items

We can delete a key-value pair using `del`, `pop()`, or `popitem()`.

#### Using `del`:

```
student = {"name": "John", "age": 20, "course": "Computer Science"}
del student["age"]
print(student) # Output: {'name': 'John', 'course': 'Computer Science'}
```

#### Using `.pop()` (Removes & Returns the Value):

```
removed_value = student.pop("course")
print(removed_value) # Output: Computer Science
print(student) # Output: {'name': 'John'}
```

#### Using `.popitem()` (Removes Last Inserted Item in Python 3.7+):

```
student = {"name": "John", "age": 20, "course": "CS"}
student.popitem()
print(student) # Output: {'name': 'John', 'age': 20}
```

#### Using `.clear()` (Removes All Items):

```
student.clear()
print(student) # Output: {}
```

### 4. Iterating Through a Dictionary

You can loop through a dictionary using a `for` loop.

#### Looping Through Keys:

```
student = {"name": "John", "age": 20, "course": "CS"}
for key in student:
```

```

print(key)
# Output:
# name
# age
# course

```

**Looping Through Values:**

```

for value in student.values():
    print(value)
# Output:
# John
# 20
# CS

```

**Looping Through Both Keys & Values (.items()):**

```

for key, value in student.items():
    print(f"{key}: {value}")
# Output:
# name: John
# age: 20
# course: CS

```

**5. Nested Dictionaries**

A nested dictionary means a dictionary inside another dictionary.

**Example:**

```

students = {
    "student1": {"name": "Alice", "age": 22},
    "student2": {"name": "Bob", "age": 21},

```

```
}
```

```
print(students["student1"]["name"]) # Output: Alice
```

### **Adding a New Entry to Nested Dictionary:**

```
students["student3"] = {"name": "Charlie", "age": 23}
```

```
print(students)
```

```
# Output:
```

```
# {'student1': {'name': 'Alice', 'age': 22},
```

```
# 'student2': {'name': 'Bob', 'age': 21},
```

```
# 'student3': {'name': 'Charlie', 'age': 23}}
```

### **Iterating Over a Nested Dictionary:**

```
for student, details in students.items():
```

```
    print(f"{student}:")
```

```
    for key, value in details.items():
```

```
        print(f" {key}: {value}")
```

```
# Output:
```

```
# student1:
```

```
# name: Alice
```

```
# age: 22
```

```
# student2:
```

```
# name: Bob
```

```
# age: 21
```

```
# student3:
```

```
# name: Charlie
```

```
# age: 23
```

## Summary

- ✓ Dictionaries store data in key-value pairs.
- ✓ Access values using `dict[key]` or `.get()`.
- ✓ Add or update values using `dict[key] = value` or `.update()`.
- ✓ Remove elements using `del`, `.pop()`, `.popitem()`, or `.clear()`.
- ✓ Loop through dictionaries using `.keys()`, `.values()`, or `.items()`.
- ✓ Nested Dictionaries allow hierarchical data storage.

## Mini Project 1: Student Database Management System

### Concepts Covered:

- ✓ Accessing Dictionary Items
- ✓ Adding and Updating Dictionary Items
- ✓ Removing Dictionary Items
- ✓ Iterating Through a Dictionary
- ✓ Nested Dictionaries

### Project Description:

Create a Student Database Management System using dictionaries. The system should allow the user to add new students, update student details, remove students, and display all students' details.

### Steps to Implement:

1. Create an empty dictionary to store student details.
2. Add students using their ID as the key and a nested dictionary with details like name, age, and course.

3. Allow the user to update student information.
4. Allow the user to remove a student from the database.
5. Provide an option to view all students with details.

**Python Code:**

```
# Student Database System
students = {}

def add_student(student_id, name, age, course):
    students[student_id] = {"name": name, "age": age, "course": course}
    print(f"Student {name} added successfully!")

def update_student(student_id, key, value):
    if student_id in students:
        students[student_id][key] = value
        print(f"Student {student_id} updated successfully!")
    else:
        print("Student not found!")

def remove_student(student_id):
    if student_id in students:
        del students[student_id]
        print(f"Student {student_id} removed successfully!")
    else:
        print("Student not found!")

def display_students():
    if students:
        for student_id, details in students.items():
            print(f"ID: {student_id}, Name: {details['name']}, Age: {details['age']},
Course: {details['course']}")
```

```
else:  
    print("No students in the database.")  
  
# Example Usage  
add_student(101, "Alice", 20, "Computer Science")  
add_student(102, "Bob", 21, "Mathematics")  
  
update_student(101, "age", 21)  
remove_student(102)  
  
display_students()
```

**Expected Output:**

Student Alice added successfully!  
Student Bob added successfully!  
Student 101 updated successfully!  
Student 102 removed successfully!  
ID: 101, Name: Alice, Age: 21, Course: Computer Science

## Mini Project 2: Inventory Management System

**Concepts Covered:**

- ✓ Accessing Dictionary Items
- ✓ Adding and Updating Dictionary Items
- ✓ Removing Dictionary Items
- ✓ Iterating Through a Dictionary
- ✓ Nested Dictionaries

**Project Description:**

Create an Inventory Management System for a store using dictionaries. The system should allow the user to add new products, update stock levels, remove products, and display available products.

**Steps to Implement:**

1. Create an empty dictionary to store product details.
2. Use Product ID as the key and store name, price, and quantity in a nested dictionary.
3. Allow the user to update the stock quantity and price.
4. Allow the user to remove a product from the inventory.
5. Provide an option to view all available products.

**Python Code:**

```
# Inventory Management System
inventory = {}

def add_product(product_id, name, price, quantity):
    inventory[product_id] = {"name": name, "price": price, "quantity": quantity}
    print(f"Product {name} added successfully!")

def update_product(product_id, key, value):
    if product_id in inventory:
        inventory[product_id][key] = value
        print(f"Product {product_id} updated successfully!")
    else:
        print("Product not found!")

def remove_product(product_id):
    if product_id in inventory:
        del inventory[product_id]
        print(f"Product {product_id} removed successfully!")
```

```
else:  
    print("Product not found!")  
  
def display_inventory():  
    if inventory:  
        for product_id, details in inventory.items():  
            print(f"ID: {product_id}, Name: {details['name']}, Price: {details['price']},  
Quantity: {details['quantity']}")  
    else:  
        print("No products in inventory.")  
  
# Example Usage  
add_product(1, "Laptop", 800, 10)  
add_product(2, "Mouse", 20, 50)  
  
update_product(1, "price", 750)  
remove_product(2)  
  
display_inventory()
```

**Expected Output:**

Product Laptop added successfully!  
Product Mouse added successfully!  
Product 1 updated successfully!  
Product 2 removed successfully!  
ID: 1, Name: Laptop, Price: 750, Quantity: 10

### **Key Takeaways from These Projects:**

- ✓ Dictionaries are powerful for storing structured data.
- ✓ Accessing dictionary items allows retrieving specific values.
- ✓ Adding and updating items makes it dynamic and flexible.
- ✓ Removing items keeps the data relevant and manageable.
- ✓ Iterating through dictionaries helps in displaying structured data.
- ✓ Nested dictionaries help organize complex data efficiently.

## **Day 31 tasks**

### **Task 1: Create a Dictionary and Access Elements**

Create a dictionary with three key-value pairs (e.g., "name": "Alice", "age": 25, "city": "New York").

- ✓ Access the values of "name" and "age" using both square brackets (dict[key]) and get() method.

### **Task 2: Handle Missing Keys While Accessing Dictionary Items**

Try to access a non-existent key from a dictionary.

- ✓ Use the get() method to avoid errors and return a default value instead.

### **Task 3: Add New Key-Value Pairs to a Dictionary**

Start with an empty dictionary and add three key-value pairs dynamically.

- ✓ Print the updated dictionary after each addition.

### **Task 4: Update an Existing Dictionary Entry**

Create a dictionary with product details ("name", "price", "stock").

- ✓ Update the "price" and "stock" values.

- ✓ Print the dictionary before and after updating.

### **Task 5: Merge Two Dictionaries**

Create two separate dictionaries.

- ✓ Merge them using the update() method.
- ✓ Print the merged dictionary.

### **Task 6: Remove a Specific Key from a Dictionary**

Create a dictionary with five key-value pairs.

- ✓ Remove a key using the del statement.
- ✓ Try removing a key that doesn't exist and handle the error properly.

### **Task 7: Remove an Item Using pop() Method**

Create a dictionary with three key-value pairs.

- ✓ Remove a specific key using pop() and store the removed value in a variable.
- ✓ Print the removed value and the updated dictionary.

### **Task 8: Remove and Return the Last Item from a Dictionary**

Create a dictionary with at least three items.

- ✓ Use the popitem() method to remove and return the last item.
- ✓ Print the removed item and the updated dictionary.

### **Task 9: Iterate Through a Dictionary (Keys & Values)**

Create a dictionary with three key-value pairs.

- ✓ Iterate through the dictionary and print each key and its value.

### **Task 10: Iterate Through a Dictionary and Extract Keys Only**

Create a dictionary and iterate through it to print only the keys.

- ✓ Use the keys() method.

**Task 11: Iterate Through a Dictionary and Extract Values Only**

Create a dictionary and iterate through it to print only the values.

- ✓ Use the values() method.

**Task 12: Iterate Through a Nested Dictionary**

Create a nested dictionary with student details (e.g., name, age, subjects).

- ✓ Write a loop to iterate through each student and print their details.

**Example:**

```
students = {
    "student1": {"name": "Alice", "age": 20, "subject": "Math"},
    "student2": {"name": "Bob", "age": 21, "subject": "Science"}
}
```

**Task 13: Access a Specific Value from a Nested Dictionary**

Use a nested dictionary and access a specific value (e.g., the subject of "student2").

- ✓ Use both direct indexing and the get() method.

**Key Takeaways from These Tasks:**

- ✓ Access dictionary items safely using get().
- ✓ Add & update dictionary items dynamically.
- ✓ Remove elements using del, pop(), and popitem().
- ✓ Iterate through dictionaries to retrieve keys, values, or both.
- ✓ Work with nested dictionaries for structured data storage.

## Mini Project 1: Contact Book Application

**Goal:** Create a contact book where users can store and manage contacts (name, phone number, email).

### Features:

1. Add a new contact (name, phone, email).
2. Update contact details (change phone number or email).
3. Delete a contact by name.
4. Search for a contact by name.
5. Display all contacts in a structured format.

### Example Dictionary Structure:

```
contacts = {  
    "John Doe": {"phone": "9876543210", "email": "john@example.com"},  
    "Alice Smith": {"phone": "9123456789", "email": "alice@example.com"},  
}
```

## Mini Project 2: Library Book Management System

**Goal:** Build a library management system that keeps track of books available in a library.

### Features:

1. Add a new book (title, author, copies available).
2. Update book availability (borrowed or returned).
3. Remove a book from the collection.
4. List all available books.
5. Use nested dictionaries to store book details.

**Example Dictionary Structure:**

```
library = {  
    "Book001": {"title": "Python Programming", "author": "Guido van Rossum",  
    "copies": 5},  
    "Book002": {"title": "Data Science Essentials", "author": "Andrew Ng", "copies":  
    3},  
}
```

## Day 32

### What is a Set in Python?

A set in Python is an unordered collection of unique elements. It does not allow duplicate values and is mutable (modifiable). However, sets do not maintain order like lists or tuples.

#### Key Features of Sets:

- ✓ Unordered – Elements have no fixed position.
- ✓ Unique – No duplicate values.
- ✓ Mutable – You can add or remove items.
- ✓ Fast operations – Searching and removing items is faster compared to lists.

#### Example of a Set:

```
my_set = {1, 2, 3, 4, 5}  
print(my_set) # Output: {1, 2, 3, 4, 5}
```

## Access Set Items

Since sets are unordered, you cannot access elements by index like lists. However, you can check if an element exists in a set.

### Example:

```
my_set = {"apple", "banana", "cherry"}  
# Checking if an item exists  
print("banana" in my_set) # Output: True
```

## Add Set Items

You can add elements using `add()` and multiple elements using `update()`.

### Example:

```
fruits = {"apple", "banana"}  
fruits.add("cherry") # Adds one item  
print(fruits) # Output: {'apple', 'banana', 'cherry'}  
  
fruits.update(["mango", "orange"]) # Adds multiple items  
print(fruits) # Output: {'apple', 'banana', 'cherry', 'mango', 'orange'}
```

## Remove Set Items

You can remove items using `remove()` or `discard()`.

- `remove()` raises an error if the item is not found.
- `discard()` does not raise an error if the item is missing.

### Example:

```
fruits = {"apple", "banana", "cherry"}  
fruits.remove("banana") # Removes 'banana'  
print(fruits) # Output: {'apple', 'cherry'}
```

```
fruits.discard("grape") # No error if 'grape' is not found
```

- To remove all elements, use clear().
- To delete the set completely, use del.

```
fruits.clear() # Empty set
del fruits # Deletes the set
```

## Join Sets

You can join two sets using union() or update().

### Example:

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
```

```
# Union creates a new set
new_set = set1.union(set2)
print(new_set) # Output: {1, 2, 3, 4, 5}
```

```
# Update modifies set1
set1.update(set2)
print(set1) # Output: {1, 2, 3, 4, 5}
```

### Other set operations:

- Intersection (&) – Common elements
- Difference (-) – Elements in set1 but not in set2
- Symmetric Difference (^) – Elements in either set but not both

## Set Methods

Some useful set methods:

Method	Description
add(x)	Adds element x to the set.
remove(x)	Removes x (raises an error if not found).
discard(x)	Removes x (no error if not found).
clear()	Removes all elements.
union(set2)	Returns a new set with elements from both sets.
intersection(set2)	Returns a set with common elements.
difference(set2)	Returns a set with elements not in set2.
symmetric_difference(set2)	Returns elements in either but not both.

## Loop Sets

Since sets are iterable, you can loop through them using a for loop.

### Example:

```
fruits = {"apple", "banana", "cherry"}
for fruit in fruits:
    print(fruit)
```

### Output (unordered):

banana  
cherry  
apple

## What is a Frozen Set?

A frozen set is an immutable set, meaning you cannot add or remove elements after creating it.

### Example:

```
frozen_set = frozenset(["apple", "banana", "cherry"])
print(frozen_set) # Output: frozenset({'apple', 'banana', 'cherry'})

# frozen_set.add("orange") # ✗ Error! Frozen sets cannot be modified.
```

### Use cases of frozenset:

- When you need an unchangeable collection of items (e.g., keys in a dictionary).
- To store unique, hashable values securely.

## Summary

- ◆ Sets store unique, unordered elements.
- ◆ You can add (add()), remove (remove()), and join (union()) sets.
- ◆ Looping is possible using a for loop.
- ◆ Frozensets are immutable versions of sets.

## Mini Project 1: Student Attendance Management System

**Concepts Used:** Access Set Items, Add Set Items, Remove Set Items, Loop Sets, Set Methods

**Project Description:**

Create a Student Attendance Management System using Python sets. This system will:

- ✓ Store the list of students present.
- ✓ Allow teachers to mark attendance (add students).
- ✓ Allow removal of absent students.
- ✓ Check if a student is present.
- ✓ Display all present students.

**Code Implementation:**

```
# Initialize an empty set for attendance
attendance = set()

# Function to mark attendance
def mark_attendance(name):
    attendance.add(name)
    print(f"{name} marked as present.")

# Function to remove a student
def remove_student(name):
    if name in attendance:
        attendance.remove(name)
        print(f"{name} marked as absent.")
    else:
        print(f"{name} is not in the attendance list.")

# Function to display attendance
```

```
def display_attendance():
    if attendance:
```

```
print("\n❖ Students Present Today:")
for student in attendance:
    print(f"- {student}")
else:
    print("\n✖ No students present.")

# Sample Execution
mark_attendance("Alice")
mark_attendance("Bob")
mark_attendance("Charlie")
display_attendance()

remove_student("Bob")
display_attendance()
```

### Expected Output

Alice marked as present.  
Bob marked as present.  
Charlie marked as present.

Students Present Today:

- Alice
- Bob
- Charlie

Bob marked as absent.

Students Present Today:

- Alice
- Charlie

## Mini Project 2: Secure Password Management System (Frozen Set)

**Concepts Used:** Frozen Set, Join Sets, Set Methods

**Project Description:**

Create a Secure Password Management System where:

- ✓ Users can store a list of strong passwords (Immutable – frozenset).
- ✓ The system will compare user input with stored passwords to check if the password is secure.
- ✓ The system can suggest alternative passwords by joining sets.

**Code Implementation:**

```
# Frozen set of strong passwords (Cannot be changed)
strong_passwords = frozenset({"P@ssw0rd123", "Secure#456", "Python$789"})

# Function to check if a password is strong
def check_password(password):
    if password in strong_passwords:
        print("✓ Strong Password!")
    else:
        print("✗ Weak Password! Consider using one of these:")
        print(strong_passwords)

# Function to generate a new set of suggested passwords
def suggest_passwords():
    additional_passwords = {"Safe@111", "Strong$222"}
    new_suggestions = strong_passwords.union(additional_passwords) # Join sets
    print("\n💡 Suggested Strong Passwords:")
    print(new_suggestions)
```

```
# Sample Execution  
check_password("P@ssw0rd123") # Strong  
check_password("weakpass")    # Weak  
suggest_passwords()
```

### Expected Output

✓ Strong Password!  
✗ Weak Password! Consider using one of these:  
`frozenset({'Secure#456', 'Python$789', 'P@ssw0rd123'})`

💡 Suggested Strong Passwords:  
`{'Secure#456', 'Safe@111', 'Python$789', 'Strong$222', 'P@ssw0rd123'}`

### Summary of Concepts Used

- ◆ Set operations (Adding, Removing, Checking, Looping) → Student Attendance System
- ◆ Frozen Sets (Immutable, Secure Storage), Join Sets → Password Management System

## Day 32 tasks

### Task 1: Create and Access a Set

Create a set of five favorite colors.

Print each color using a loop.

### Task 2: Add Items to a Set

Create an empty set.

Add five favorite movies to the set.

Print the updated set.

### **Task 3: Remove Items from a Set**

Create a set of six fruits.

Remove a fruit using remove().

Try to remove a non-existing fruit using discard().

Print the final set.

### **Task 4: Check if an Item Exists in a Set**

Create a set of programming languages.

Ask the user to enter a language.

Check if the language exists in the set and display the result.

### **Task 5: Join Two Sets**

Create two sets:

1. Even numbers up to 10

2. Odd numbers up to 10

Join both sets using union() and print the result.

### **Task 6: Find the Common Elements in Two Sets**

Create two sets:

Set 1: {2, 4, 6, 8, 10}

Set 2: {4, 8, 12, 16}

Find the common elements using intersection() and print them.

### **Task 7: Find the Difference Between Two Sets**

Create two sets:

Set A: {1, 2, 3, 4, 5, 6}

Set B: {4, 5, 6, 7, 8, 9}

Find the difference (A - B) using difference() and print the result.

**Task 8: Symmetric Difference Between Two Sets**

Create two sets with some common values.

Find the symmetric difference (elements that are in either of the sets but not in both).

Print the result.

**Task 9: Loop Through a Set**

Create a set of four car brands.

Use a loop to print each brand name.

**Task 10: Convert a List to a Set**

Create a list of numbers with duplicates.

Convert it to a set to remove duplicates.

Print the unique numbers.

**Task 11: Frozen Set Example**

Create a frozen set of vowels {'a', 'e', 'i', 'o', 'u'}.

Try to add an element and observe what happens.

**Task 12: Perform Set Operations on a Frozen Set**

Create a frozen set of prime numbers up to 10.

Create another set of even numbers up to 10.

Try performing intersection and union.

**Task 13: Find the Length of a Set**

Create a set of 10 random words.

Use the len() function to find and print the number of items in the set.

## Mini Project 1: Student Course Enrollment System

### Concepts Used:

- ✓ Accessing, Adding, Removing Set Items
- ✓ Using Set Methods (add(), remove(), discard(), union())
- ✓ Looping through a Set

### Project Requirements:

1. Create a set available\_courses containing 5 different course names.
2. Allow the user to enroll in multiple courses by adding them to a student\_courses set.
3. If the course is not in available\_courses, display "Course not found!".
4. Allow the user to remove a course if they change their mind.
5. Show the final list of enrolled courses at the end.

### Example Output:

```
Available Courses: {'Python', 'Java', 'C++', 'Web Development', 'Data Science'}
Enter a course to enroll: Python
Enter another course to enroll: Java
Enter a course to remove (if any): Java
Final Enrolled Courses: {'Python'}
```

## Mini Project 2: Unique Word Counter from a Paragraph

### Concepts Used:

- ✓ Accessing, Adding, Removing Set Items
- ✓ Using Set Methods (add(), len(), intersection(), difference())
- ✓ Looping through a Set
- ✓ Using Frozen Sets

**Project Requirements:**

1. Ask the user to input a paragraph.
2. Convert the paragraph into a set of unique words (ignore case sensitivity).
3. Store common words like {"is", "a", "the", "and", "to", "of", "in"} in a frozen set.
4. Remove common words from the unique words set.
5. Display the total unique words and print them.

**Example Output:**

Enter a paragraph: Python is a powerful programming language. Python helps in automation.

Unique Words: {'powerful', 'helps', 'programming', 'automation', 'language', 'Python'}

Total Unique Words (excluding common words): 6

## Python Advanced Concepts

### Day 33

#### Python Modules for Data Analytics

##### Introduction to Python Modules

A module in Python is simply a file containing Python code, which can include functions, classes, and variables. Modules help in code reusability and organization.

## Types of Modules in Python

Python has three types of modules:

1. Built-in Modules – Pre-installed with Python (e.g., math, random, os).
2. User-Defined Modules – Custom modules created by the user.
3. External Modules – Installed using pip (e.g., pandas, numpy).

## Using Built-in Modules

Python comes with many built-in modules that provide various functionalities.

### Example 1: Using the math Module

```
import math
```

```
print(math.sqrt(16)) # Square root  
print(math.factorial(5)) # Factorial  
print(math.pi) # Value of pi
```

### Example 2: Using the random Module

```
import random
```

```
print(random.randint(1, 10)) # Random integer between 1 and 10  
print(random.choice(['apple', 'banana', 'cherry'])) # Random choice from a list
```

## Creating a User-Defined Module

You can create your own module by saving a Python file (.py) with functions and variables.

**Step 1: Create a Module (e.g., mymodule.py)**

```
# mymodule.py
def greet(name):
    return f"Hello, {name}!"

pi_value = 3.14159
```

**Step 2: Import the Module in Another File**

```
import mymodule

print(mymodule.greet("John"))
print(mymodule.pi_value)
```

**Step 3: Import Only Specific Functions**

```
from mymodule import greet

print(greet("Alice"))
```

**Using External Modules (Third-Party)**

External modules are not included in Python by default. They must be installed using pip.

**Example: Install and Use pandas**

```
pip install pandas
import pandas as pd
```

```
data = {"Name": ["Alice", "Bob"], "Age": [25, 30]}
df = pd.DataFrame(data)
print(df)
```

## Different Ways to Import Modules

### 1. Import the Whole Module

```
import math  
print(math.sqrt(25))
```

### 2. Import a Specific Function

```
from math import sqrt  
print(sqrt(25))
```

### 3. Import with an Alias

```
import math as m  
print(m.sqrt(25))
```

### 4. Import All Functions (Not Recommended)

```
from math import *  
print(sqrt(25)) # No need to use math.sqrt()
```

## Finding Available Functions in a Module

Use the `dir()` function to list all functions and variables inside a module.

```
import math  
print(dir(math))
```

## Check Module Documentation

Use the `help()` function to get information about a module.

```
import math  
help(math)
```

## Summary

- ✓ Built-in Modules (e.g., math, random, os)
- ✓ User-Defined Modules (.py file created by the user)
- ✓ Third-Party Modules (pip install module\_name)
- ✓ Ways to Import Modules (import, from ... import ...)
- ✓ Finding Functions (dir(), help())

## Next Steps

- Try importing different built-in modules (os, datetime, sys).
- Create and import your own module.
- Use a third-party module (pandas, numpy).

## Introduction to Python Packages

A package in Python is a collection of modules (Python files) organized in directories. Packages help structure large applications and enable code reuse.

## Difference Between Modules and Packages

Feature	Module	Package
Definition	A single Python file (.py)	A collection of modules in a directory
Structure	Contains functions, classes, and variables	Contains multiple modules and a special <code>__init__.py</code> file
Example	<code>math.py, random.py</code>	<code>pandas, numpy, matplotlib</code>

## Creating a Python Package

A package is a directory that contains multiple modules and a special file called `__init__.py`.

**Step 1: Create a Package Structure**

```
mypackage/  
|__ __init__.py    # Marks the directory as a package  
|__ math_utils.py # Module for math operations  
|__ string_utils.py # Module for string functions
```

**Step 2: Create Modules in the Package****math\_utils.py**

```
def add(a, b):  
    return a + b
```

```
def subtract(a, b):  
    return a - b
```

**string\_utils.py**

```
def to_uppercase(text):  
    return text.upper()
```

```
def to_lowercase(text):  
    return text.lower()
```

**Importing from a Package**

After creating a package, you can **import modules** in different ways.

**1. Import the Entire Module**

```
import mypackage.math_utils
```

```
result = mypackage.math_utils.add(10, 5)  
print(result) # Output: 15
```

## 2. Import Specific Functions

```
from mypackage.math_utils import add
```

```
print(add(10, 5)) # Output: 15
```

## 3. Import the Whole Package

To allow import mypackage, modify `__init__.py`:

```
from .math_utils import add, subtract  
from .string_utils import to_uppercase, to_lowercase
```

Now, you can import the package directly:

```
import mypackage
```

```
print(mypackage.add(10, 5))  
print(mypackage.to_uppercase("hello"))
```

## Installing and Using External Packages

Python provides thousands of pre-built packages via PyPI (Python Package Index).

### What is pip in Python?

pip (Package Installer for Python) is the default package manager for Python. It allows you to:

- ✓ Install Python libraries
- ✓ Upgrade installed packages
- ✓ Uninstall packages
- ✓ List installed packages

## Checking if pip is Installed

By default, pip comes with Python (version 3.4 and later). To check if pip is installed, run:

```
pip --version
```

### Example Output:

```
pip 22.0.2 from C:\Python\Lib\site-packages\pip (python 3.10)
```

## Installing a Package with pip

You can install **any package** from PyPI (Python Package Index) using:

```
pip install package_name
```

### Example: Installing pandas

```
pip install pandas
```

## Installing a Specific Package Version

To install a **specific version** of a package, use:

```
pip install package_name==version
```

### Example: Install numpy version 1.21.0

```
pip install numpy==1.21.0
```

## Upgrading a Package

To upgrade an existing package to the latest version, use:

```
pip install --upgrade package_name
```

### Example: Upgrade matplotlib

```
pip install --upgrade matplotlib
```

## Listing Installed Packages

To see all installed packages, use:

```
pip list
```

### Example Output:

numpy	1.21.0
pandas	1.3.3
matplotlib	3.4.3

To check details of a specific package, use:

```
pip show package_name
```

### Example:

```
pip show pandas
```

## Uninstalling a Package

To remove a package, use:

```
pip uninstall package_name
```

### **Example: Uninstall seaborn**

```
pip uninstall seaborn
```

## **Installing Multiple Packages from a File**

You can create a requirements file to install multiple packages at once.

### **Step 1: Create requirements.txt**

```
numpy==1.21.0  
pandas==1.3.3  
matplotlib
```

### **Step 2: Install all Packages from requirements.txt**

```
pip install -r requirements.txt
```

## **Installing Packages in a Virtual Environment**

To prevent conflicts between different Python projects, use a virtual environment.

### **Step 1: Create a Virtual Environment**

```
python -m venv myenv
```

### **Step 2: Activate the Virtual Environment**

- Windows: myenv\Scripts\activate
- Mac/Linux: source myenv/bin/activate

### **Step 3: Install Packages Inside the Virtual Environment**

```
pip install numpy pandas
```

## Step 4: Deactivate the Virtual Environment

deactivate

### Summary

- ✓ A package is a directory containing multiple modules
- ✓ It must include an `__init__.py` file
- ✓ You can import modules or specific functions from a package
- ✓ External packages are installed using pip

Command	Description
<code>pip --version</code>	Check pip version
<code>pip install package_name</code>	Install a package
<code>pip install package_name==version</code>	Install a specific version
<code>pip install --upgrade package_name</code>	Upgrade a package
<code>pip list</code>	List installed packages
<code>pip show package_name</code>	Show package details
<code>pip uninstall package_name</code>	Remove a package
<code>pip install -r requirements.txt</code>	Install multiple packages

### Next Steps

1. Create your own Python package and test imports.
2. Try installing and using external packages (pandas, numpy, matplotlib).
3. Try installing, upgrading, and uninstalling packages using pip.
4. Use pip list to explore installed packages in your system.

## Python Modules for Data Analytics

In Data Analytics, Python provides powerful modules to analyze, process, and visualize data. Here, we will focus on the most important built-in and external modules used in data analytics.

### Essential Python Modules for Data Analytics

Module	Purpose
pandas	Data manipulation and analysis
numpy	Numerical computations
matplotlib	Data visualization
seaborn	Statistical data visualization
scipy	Scientific computing
statsmodels	Statistical modeling and hypothesis testing
sklearn	Machine learning for analytics

## Mini Projects

### Mini Project 1. Using pandas for Data Handling

The pandas module is used to read, process, and analyze structured data.

#### Install pandas (if not installed)

pip install pandas

#### Example 1: Creating and Displaying a DataFrame

```
import pandas as pd
```

```
# Creating a dataset
data = {
    "Name": ["Alice", "Bob", "Charlie"],
    "Age": [25, 30, 35],
```

```
"Salary": [50000, 60000, 70000]  
}
```

```
df = pd.DataFrame(data)  
print(df)
```

### **Example 2: Reading a CSV File**

```
df = pd.read_csv("data.csv") # Load dataset  
print(df.head()) # View first 5 rows
```

## **Mini Project 2. Using numpy for Numerical Computations**

The numpy module is used for **handling arrays and numerical operations**.

### **Install numpy**

```
pip install numpy
```

### **Example 1: Creating and Manipulating Arrays**

```
import numpy as np  
  
arr = np.array([10, 20, 30, 40])  
print("Array:", arr)  
print("Mean:", np.mean(arr)) # Calculate Mean  
print("Standard Deviation:", np.std(arr)) # Standard deviation
```

## **Mini Project 3. Using matplotlib for Data Visualization**

The matplotlib module is used for plotting graphs and charts.

### Install matplotlib

```
pip install matplotlib
```

### Example: Plot a Simple Line Chart

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [10, 20, 30, 40, 50]

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

### Mini Project 4. Using seaborn for Statistical Visualization

The seaborn module helps create beautiful statistical graphs.

### Install seaborn

```
pip install seaborn
```

### Example: Plot a Histogram

```
import seaborn as sns
import pandas as pd

import matplotlib.pyplot as plt

# Sample dataset
data = {"Age": [22, 25, 30, 35, 40, 45, 50]}
df = pd.DataFrame(data)
```

```
sns.histplot(df["Age"], bins=5, kde=True)
plt.show()
```

## Mini Project 5. Using scipy for Scientific Computation

The scipy module provides scientific computing and statistics.

### Install scipy

```
pip install scipy
```

### Example: Finding Statistical Measures

```
from scipy import stats
import numpy as np

data = [10, 20, 30, 30, 40, 50]

print("Mean:", np.mean(data))
print("Median:", np.median(data))

# Fix: Directly access mode without indexing
mode_result = stats.mode(data, keepdims=True) # Ensure compatibility with all
versions
print("Mode:", mode_result.mode.item()) # Use .item() for scalar values
```

## Mini Project 6. Using statsmodels for Statistical Analysis

The statsmodels module is used for hypothesis testing and statistical modeling.

### Install statsmodels

```
pip install statsmodels
```

**Example: Linear Regression**

```
import statsmodels.api as sm
```

```
x = [1, 2, 3, 4, 5]
```

```
y = [10, 15, 20, 25, 30]
```

```
X = sm.add_constant(x) # Add intercept
```

```
model = sm.OLS(y, X).fit()
```

```
print(model.summary()) # View regression results
```

**Mini Project 7. Using sklearn for Machine Learning in Analytics**

The sklearn module provides machine learning tools for predictive analytics.

**Install sklearn**

```
pip install scikit-learn
```

**Example: Simple Linear Regression**

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

```
X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)
```

```
y = np.array([10, 20, 30, 40, 50])
```

```
model = LinearRegression()
```

```
model.fit(X, y)
```

```
print("Predicted Value for 6:", model.predict([[6]]))
```

## Summary

Module	Purpose
pandas	Data manipulation
numpy	Numerical calculations
matplotlib	Data visualization
seaborn	Statistical visualization
scipy	Scientific computation
statsmodels	Statistical modeling
sklearn	Machine learning

## Day 33 Tasks

1. Create a module named `math_utils.py` and define functions for addition, subtraction, multiplication, and division. Import and use this module in another script.
2. Use the built-in `math` module to calculate the square root, factorial, and power of given numbers.
3. Import only specific functions from the `random` module and generate a random number, shuffle a list, and choose a random item from a list.
4. Use the `datetime` module to display the current date, time, and day of the week.
5. Write a script that imports the `os` module and retrieves the current working directory, lists files in a directory, and creates a new folder.
6. Create a custom module named `string_utils.py` with functions for reversing a string, converting to uppercase, and counting vowels. Import and test it in another script.
7. Use the `sys` module to read command-line arguments and print them. (Hint: `sys.argv[]`)
8. Use the `time` module to measure the execution time of a Python function.

9. Create and import a module for managing student records, which includes adding a student, removing a student, and displaying student details.
10. Use the calendar module to display the calendar of a specific month and year entered by the user.
11. Use the json module to convert a Python dictionary into JSON format and save it to a file, then read it back.
12. Create a module named file\_manager.py to handle file operations like reading, writing, and appending to a file. Import and use this module in another script.
13. Use the requests module (install if needed) to fetch data from an API, such as retrieving weather information from an online API.

## **Mini Project 1: Personal Finance Tracker (Using Custom and Built-in Modules)**

### **Project Overview**

Create a Personal Finance Tracker using a custom module to manage income and expenses. The program should:

- ✓ Allow users to add income and expenses.
- ✓ Display a summary of their financial status.
- ✓ Store data in a file using the json module.

### **Key Python Modules Used**

Custom Module (finance.py) – Contains functions to add, remove, and calculate balance.

json Module – To store and retrieve finance data.

datetime Module – To timestamp transactions.

### Task Breakdown

1. Create a finance.py module with functions:

- add\_income(amount, description)
- add\_expense(amount, description)
- get\_balance()
- save\_to\_file()
- load\_from\_file()

2. Create a main script to import the finance module and allow users to input transactions.

3. Store transactions in a JSON file and retrieve them when the program starts.

4. Display the current balance and transaction history when requested.

## Mini Project 2: Weather Information App (Using Built-in and External Modules)

### Project Overview

Develop a Weather Information App that fetches the current weather of a given city using an API request.

### Key Python Modules Used

- requests Module – To fetch weather data from an API.
- json Module – To process API response data.
- sys Module – To allow command-line input of the city name.

### Task Breakdown

1. Install the requests module (pip install requests).

2. Create a function in a module (weather.py) to fetch weather data from an API like OpenWeatherMap.

3. Use the sys module to take city name input from the command line.
4. Format and display the temperature, humidity, and weather condition.
5. Allow users to save the fetched weather data to a file for later reference.

## Day 34

### Object-Oriented Programming (OOPs) in Python

Object-Oriented Programming (OOP) is a programming paradigm that uses objects and classes to structure code in a modular and reusable way. It helps in organizing code by bundling attributes (data) and behaviors (methods) into objects.

#### Key OOP Concepts in Python:

1. Class & Object
2. Encapsulation
3. Abstraction
4. Inheritance
5. Polymorphism

#### 1. Class and Object

##### Definition:

A class is a blueprint for creating objects. It defines attributes (variables) and methods (functions). An object is an instance of a class.

##### Syntax:

```
class Car: # Defining a class  
    def __init__(self, brand, model):
```

```

self.brand = brand # Attribute
self.model = model # Attribute

def display(self): # Method
    print(f"Car: {self.brand}, Model: {self.model}")

# Creating objects
car1 = Car("Toyota", "Camry")
car2 = Car("Honda", "Civic")

car1.display() # Output: Car: Toyota, Model: Camry
car2.display() # Output: Car: Honda, Model: Civic

```

- `__init__` is a special constructor method used to initialize object attributes.

## 2. Encapsulation

### Definition:

Encapsulation is data hiding. It restricts direct access to some variables and allows controlled access using getter and setter methods.

### Syntax:

`class BankAccount:`

```

def __init__(self, balance):
    self.__balance = balance # Private variable (cannot be accessed directly)

```

```

def deposit(self, amount):
    self.__balance += amount

```

```

def withdraw(self, amount):

```

```

if self.__balance >= amount:
    self.__balance -= amount
else:
    print("Insufficient funds")

def get_balance(self): # Getter method
    return self.__balance

# Creating an object
account = BankAccount(1000)
account.deposit(500)
print(account.get_balance()) # Output: 1500
account.withdraw(700)
print(account.get_balance()) # Output: 800

```

- Private variables (`__balance`) are hidden and cannot be accessed directly.

### 3. Abstraction

#### **Definition:**

Abstraction hides complex implementation details and only exposes the necessary functionalities to the user.

#### **Syntax using ABC Module:**

```

from abc import ABC, abstractmethod # Importing abstract class module

class Animal(ABC): # Abstract class
    @abstractmethod
    def sound(self): # Abstract method
        pass

```

```

class Dog(Animal):
    def sound(self):
        return "Barks"

class Cat(Animal):
    def sound(self):
        return "Meows"

# Creating objects
dog = Dog()
cat = Cat()

print(dog.sound()) # Output: Barks
print(cat.sound()) # Output: Meows

```

- `@abstractmethod` ensures that child classes must implement the method.

## 4. Inheritance

### Definition:

Inheritance allows a class (child) to inherit properties and behaviors from another class (parent). This promotes code reusability.

### Syntax:

```

class Animal: # Parent class
    def __init__(self, name):
        self.name = name

    def speak(self):
        return "Some sound"

```

```

class Dog(Animal): # Child class inheriting Animal
    def speak(self):
        return "Barks"

class Cat(Animal): # Another child class
    def speak(self):
        return "Meows"

dog = Dog("Buddy")
cat = Cat("Whiskers")

print(dog.name, ":", dog.speak()) # Output: Buddy : Barks
print(cat.name, ":", cat.speak()) # Output: Whiskers : Meows

```

- The child class overrides the parent class's method using method overriding.

## 5. Polymorphism

### Definition:

Polymorphism allows different classes to use the same method name but perform different behaviors.

### Syntax:

```

class Bird:
    def fly(self):
        return "Birds can fly"

class Eagle(Bird):
    def fly(self):
        return "Eagles fly high"

```

```
class Penguin(Bird):
    def fly(self):
        return "Penguins cannot fly"
```

```
# Polymorphism in action
birds = [Eagle(), Penguin()]
```

```
for bird in birds:
    print(bird.fly())
```

```
# Output:
# Eagles fly high
# Penguins cannot fly
```

- Method overriding in different classes allows dynamic behavior.

### Summary Table of OOPs Concepts:

Concept	Definition	Example
Class & Object	Blueprint & instance creation	Car("Toyota", "Camry")
Encapsulation	Data hiding using private variables	__balance in BankAccount
Abstraction	Hiding details and exposing only functionality	@abstractmethod in Animal class
Inheritance	Child class inheriting parent class properties	Dog(Animal)
Polymorphism	Same method, different behavior	fly() method in Eagle and Penguin

## 1. Mini Project: Library Management System

### Project Description:

A Library Management System that allows users to:

- ✓ Add books to the library 
- ✓ Borrow books (if available)
- ✓ Return books after borrowing
- ✓ View available books

### Key OOP Concepts Used:

- Class & Objects (Library, Book, User)
- Encapsulation (Hiding book details)
- Inheritance (Extending features for different users)

### Code Implementation:

```
class Library:
    def __init__(self):
        self.books = [] # List to store books

    def add_book(self, book_name):
        self.books.append(book_name)
        print(f"Book '{book_name}' added to the library!")

    def show_books(self):
        if not self.books:
            print("No books available.")
        else:
            print("Available Books:", ", ".join(self.books))

    def borrow_book(self, book_name):
```

```
if book_name in self.books:  
    self.books.remove(book_name)  
    print(f"You borrowed '{book_name}'.")  
else:  
    print("Book not available.")  
  
def return_book(self, book_name):  
    self.books.append(book_name)  
    print(f"You returned '{book_name}'.")  
  
# Create Library object  
my_library = Library()  
my_library.add_book("Python Basics")  
my_library.add_book("Data Structures")  
  
my_library.show_books()  
my_library.borrow_book("Python Basics")  
my_library.show_books()  
my_library.return_book("Python Basics")  
my_library.show_books()
```

**Expected Output:**

Book 'Python Basics' added to the library!  
Book 'Data Structures' added to the library!  
Available Books: Python Basics, Data Structures  
You borrowed 'Python Basics'.  
Available Books: Data Structures  
You returned 'Python Basics'.  
Available Books: Data Structures, Python Basics

## 2. Mini Project: ATM System

### Project Description:

An ATM System where users can:

- ✓ Check balance 
- ✓ Deposit money 
- ✓ Withdraw money 

### Key OOP Concepts Used:

- Encapsulation (Hiding bank balance)
- Abstraction (Only exposing required functions)
- Class & Objects

### Code Implementation:

```
class ATM:
    def __init__(self, balance=0):
        self.__balance = balance # Private attribute

    def check_balance(self):
        print(f"Your balance is: ${self.__balance}")

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"${amount} deposited successfully!")
        else:
            print("Deposit amount must be positive.")

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
```

```
print(f"${amount} withdrawn successfully!")
else:
    print("Insufficient balance or invalid amount.")

# Create ATM object
user1 = ATM(1000)

user1.check_balance()
user1.deposit(500)
user1.check_balance()
user1.withdraw(300)
user1.check_balance()
user1.withdraw(1500) # Should display insufficient balance
```

**Expected Output:**

Your balance is: \$1000  
\$500 deposited successfully!  
Your balance is: \$1500  
\$300 withdrawn successfully!  
Your balance is: \$1200  
Insufficient balance or invalid amount.

**Benefits of These Projects:**

- ✓ Hands-on practice with real-world applications
- ✓ Understanding OOP principles in action
- ✓ Learning Encapsulation, Inheritance, and Abstraction

## Day 34 Tasks:

### Task 1: Create a Class and Object

Create a class Car with attributes like brand, model, and year.

Create an object of this class and print its details.

### Task 2: Add Methods to a Class

Add a method start\_engine() in the Car class that prints "Engine started!".

Call this method using an object.

### Task 3: Constructor (init Method)

Modify the Car class to initialize values using a constructor.

Create multiple car objects with different attributes.

### Task 4: Encapsulation (Private Variables)

Create a class BankAccount with a private attribute \_\_balance.

Add methods to deposit, withdraw, and check balance while keeping \_\_balance private.

### Task 5: Inheritance (Parent & Child Classes)

Create a Vehicle class with a show\_details() method.

Create a Car class that inherits from Vehicle and has additional attributes like seats.

### Task 6: Method Overriding (Polymorphism)

Override the show\_details() method in the Car class to print more specific information.

### Task 7: Multiple Inheritance

Create a Teacher class and a Researcher class, both having a method work().

Create a Professor class that inherits from both and demonstrates multiple inheritance.

### **Task 8: Abstract Class & Method**

Create an abstract class Shape with an abstract method area().

Implement subclasses Circle and Rectangle that define the area() method.

### **Task 9: Operator Overloading**

Overload the + operator in a Vector class so that adding two vectors returns a new vector.

### **Task 10: Class Method & Static Method**

Create a Person class with:

- ✓ A class method count\_people() that tracks the number of Person objects created.
- ✓ A static method is\_adult(age) that returns True if age  $\geq 18$ .

### **Task 11: File Handling with OOP**

Modify the BankAccount class to store transactions in a file (transactions.txt).

Implement save transactions and read transaction history using file handling.

### **Task 12: Mini Project - Student Management System**

Create a Student class with attributes name, age, and marks.

Implement methods to:

- ✓ Add student details
- ✓ Update student marks
- ✓ Display student details

### **Task 13: Mini Project - Employee Payroll System**

Create an Employee class with attributes like name, id, salary.

Implement methods to:

✓ Calculate Salary after tax deductions

✓ Give a raise

✓ Store employee details in a file

#### Bonus Challenge:

- Modify any of the tasks above to include exception handling (e.g., handling invalid withdrawals in BankAccount).

## 1. Mini Project Task: Hospital Management System

#### Task Description:

Create a Hospital Management System where:

- ✓ Doctors can be added with name, specialization, and available timings.
- ✓ Patients can register with name, age, and disease description.
- ✓ A patient can book an appointment with a doctor.
- ✓ The system shows all doctors and appointments.

#### Key OOP Concepts Used:

- Encapsulation (Hiding patient details)
- Inheritance (Doctor and Patient from a common Person class)
- Polymorphism (Different ways of displaying information)

#### Expected Features:

- Doctor class with attributes (name, specialization, timing)
- Patient class with attributes (name, age, disease)
- Hospital class to add doctors, register patients, and schedule appointments

## 2. Mini Project Task: Inventory Management System

### Task Description:

Create an Inventory Management System for a store where:

- ✓ Products can be added with a name, price, and quantity.
- ✓ Products can be purchased, decreasing their quantity.
- ✓ The system should show available stock and total earnings.

### Key OOP Concepts Used:

- Encapsulation (Hiding inventory details)
- Abstraction (Only exposing necessary functions)
- Method Overriding (Updating stock after purchase)

### Expected Features:

- Product class with name, price, quantity
- Store class to add products, display stock, and process sales

## Day 35

### Python File Handling

File handling in Python allows us to work with files (read, write, append, and modify files) using built-in functions. Python provides a simple and efficient way to handle files using the `open()` function.

#### 1. Opening a File

In Python, the `open()` function is used to open a file. It takes two arguments:

1. **File name** (with the path if needed)
2. **Mode** (specifies the operation to perform)

## Syntax

```
file = open("filename.txt", "mode")
```

Mode	Description
"r"	Read mode (default). Opens the file for reading. If the file does not exist, it gives an error.
"w"	Write mode. Creates a new file or overwrites an existing file.
"a"	Append mode. Adds data at the end of an existing file.
"x"	Create mode. Creates a file but gives an error if the file already exists.
"t"	Text mode (default). Opens a file in text format.
"b"	Binary mode. Opens a file in binary format (e.g., images, audio, etc.).

## 2. Reading a File

We can read the content of a file using .read(), .readline(), or .readlines() methods.

### Example: Reading a file

```
# Open the file in read mode
file = open("example.txt", "r")
```

```
# Read the entire content
content = file.read()
print(content)
```

```
# Close the file
file.close()
```

Reading Line by Line  

```
file = open("example.txt", "r")
```

```
# Read one line at a time
```

```
line = file.readline()
```

```
print(line)
```

```
file.close()
```

Reading All Lines as a List

```
file = open("example.txt", "r")
```

```
# Read all lines and store them in a list
```

```
lines = file.readlines()
```

```
print(lines)
```

```
file.close()
```

### 3. Writing to a File

To write data to a file, we use "w" mode. If the file exists, it will be overwritten.

#### Example: Writing to a File

```
file = open("example.txt", "w")
```

```
file.write("Hello, this is a new file!\n")
```

```
file.write("Python makes file handling easy.")
```

```
file.close()
```

### 4. Appending to a File

To add data to an existing file without deleting previous content, use "a" mode.

### **Example: Appending to a File**

```
file = open("example.txt", "a")
file.write("\nThis line is added using append mode.")
file.close()
```

## **5. Using with Statement (Best Practice)**

Using with automatically closes the file after the operations, reducing the risk of errors.

### **Example: Reading a File Using with**

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content) # File is automatically closed after this block
```

### **Example: Writing Using with**

```
with open("example.txt", "w") as file:
    file.write("This is a new content added safely.")
```

## **6. Working with Binary Files**

Binary mode ("b") is used to handle non-text files like images, audio, and video.

### **Example: Copying an Image File**

```
with open("image.jpg", "rb") as source:
    with open("copy.jpg", "wb") as destination:
        destination.write(source.read())
```

## **7. Checking If a File Exists (Before Opening)**

To avoid errors while opening a file, we can check if it exists using the os module.

```
import os

if os.path.exists("example.txt"):
    with open("example.txt", "r") as file:
        print(file.read())
else:
    print("File does not exist!")
```

## Summary

- ✓ Use `open("filename", "mode")` to handle files.
- ✓ Always close the file using `.close()` or with `open()`.
- ✓ "r" mode for reading, "w" for writing, "a" for appending, "x" for creating.
- ✓ Use "b" mode for binary files.

## 1. Mini Project: Simple To-Do List (Store Tasks in a File)

### Project Overview

This project allows users to add, view, and remove tasks, storing them in a file (`tasks.txt`). Every time the program runs, it loads the existing tasks from the file.

### Features

- ✓ Add tasks
- ✓ View tasks
- ✓ Remove a task
- ✓ Save tasks in a file

**Code**

```
# To-Do List using File Handling
```

```
def display_tasks():
    """Displays all tasks from the file."""
    try:
        with open("tasks.txt", "r") as file:
            tasks = file.readlines()
            if not tasks:
                print("\nNo tasks available.")
            else:
                print("\nYour Tasks:")
                for index, task in enumerate(tasks, start=1):
                    print(f"{index}. {task.strip()}")
    except FileNotFoundError:
        print("\nNo tasks found. Start by adding new tasks!")

def add_task():
    """Adds a new task to the file."""
    task = input("Enter a new task: ")
    with open("tasks.txt", "a") as file:
        file.write(task + "\n")
    print("Task added successfully!")

def remove_task():
    """Removes a task from the file."""
    display_tasks()
    try:
        with open("tasks.txt", "r") as file:
            tasks = file.readlines()
```

```
if not tasks:  
    return  
  
task_num = int(input("\nEnter the task number to remove: "))  
if 1 <= task_num <= len(tasks):  
    del tasks[task_num - 1]  
    with open("tasks.txt", "w") as file:  
        file.writelines(tasks)  
    print("Task removed successfully!")  
else:  
    print("Invalid task number!")  
except ValueError:  
    print("Please enter a valid number.")  
  
# Main program loop  
while True:  
    print("\nTo-Do List")  
    print("1. View Tasks")  
    print("2. Add Task")  
    print("3. Remove Task")  
    print("4. Exit")  
  
    choice = input("Enter your choice: ")  
  
    if choice == "1":  
        display_tasks()  
    elif choice == "2":  
        add_task()  
    elif choice == "3":  
        remove_task()  
    elif choice == "4":
```

```

print("Goodbye!")
break
else:
    print("Invalid choice. Please try again.")

```

## 2. Mini Project: Student Record Manager (CSV File)

### Project Overview

This project allows users to add, view, and search student records in a file (students.csv). It demonstrates how to work with structured data using files.

### Features

- ✓ Add student details (Name, Age, Grade)
- ✓ View all students
- ✓ Search for a student

### Code

```

import csv

FILENAME = "students.csv"

def add_student():
    """Adds a new student to the CSV file."""
    name = input("Enter student name: ")
    age = input("Enter student age: ")
    grade = input("Enter student grade: ")

    with open(FILENAME, "a", newline="") as file:
        writer = csv.writer(file)
        writer.writerow([name, age, grade])

```

```
print("Student record added successfully!")

def view_students():
    """Displays all students from the CSV file."""
    try:
        with open(FILENAME, "r") as file:
            reader = csv.reader(file)
            students = list(reader)

        if not students:
            print("\nNo student records found.")
            return

        print("\nStudent Records:")
        print(f"{'Name':<15}{{'Age':<5}{{'Grade'}}")
        print("-" * 30)
        for student in students:
            print(f"{student[0]:<15}{student[1]:<5}{student[2]}")
    except FileNotFoundError:
        print("\nNo student records found.")

def search_student():
    """Searches for a student by name."""
    search_name = input("Enter student name to search: ").strip().lower()

    try:
        with open(FILENAME, "r") as file:
            reader = csv.reader(file)
            found = False
            for student in reader:
                if student[0].strip().lower() == search_name:
                    found = True
                    break
            if not found:
                print("Student not found")
    except FileNotFoundError:
        print("File not found")
```

```
    print(f"\nFound: Name: {student[0]}, Age: {student[1]}, Grade:  
{student[2]}")  
        found = True  
        break  
    if not found:  
        print("Student not found!")  
except FileNotFoundError:  
    print("\nNo student records found.")  
  
# Main program loop  
while True:  
    print("\nStudent Record Manager")  
    print("1. Add Student")  
    print("2. View Students")  
    print("3. Search Student")  
    print("4. Exit")  
  
    choice = input("Enter your choice: ")  
  
    if choice == "1":  
        add_student()  
    elif choice == "2":  
        view_students()  
    elif choice == "3":  
        search_student()  
    elif choice == "4":  
        print("Goodbye!")  
        break  
    else:  
        print("Invalid choice. Please try again.")
```

## Summary

1. To-Do List: Stores tasks in a text file (tasks.txt). You can add, view, and remove tasks.
2. Student Record Manager: Uses a CSV file (students.csv) to store structured data and allows adding, viewing, and searching students.

## Day 35 Tasks

### 1. Create and Write to a File

Task: Create a file called data.txt and write "Hello, File Handling!" into it.

Concept: Writing to a file using "w" mode.

### 2. Read a File and Display Content

Task: Open data.txt and display its content on the console.

Concept: Reading a file using "r" mode.

### 3. Append Data to an Existing File

Task: Add "This is a new line." to data.txt without overwriting existing content.

Concept: Appending data using "a" mode.

### 4. Count the Number of Words in a File

Task: Read a file and count the total number of words in it.

Concept: Reading file content and using split() for word counting.

### 5. Copy Contents from One File to Another

Task: Copy all content from data.txt into a new file called copy.txt.

Concept: Using read() and write() together.

## **6. Read a File Line by Line**

Task: Read a file and display each line separately.

Concept: Using a for loop to iterate through lines.

## **7. Search for a Word in a File**

Task: Ask the user for a word and check if it exists in data.txt.

Concept: Using in keyword and file handling.

## **8. Replace a Word in a File**

Task: Find and replace the word "old" with "new" in data.txt.

Concept: Reading file content, modifying it, and writing back.

## **9. Store and Retrieve a List in a File**

Task: Store a list of names in names.txt and later retrieve them.

Concept: Writing and reading lists using join() and split().

## **10. Count the Number of Lines in a File**

Task: Count and display the total number of lines in data.txt.

Concept: Using readlines() and len().

## **11. Merge Two Files into One**

Task: Merge file1.txt and file2.txt into a new file merged.txt.

Concept: Reading multiple files and writing content into one file.

## **12. Work with CSV Files**

Task: Create a CSV file (students.csv) and store student names with their scores.

Then, read and display the records.

Concept: Using the csv module to write and read structured data.

### 13. Remove Blank Lines from a File

Task: Remove empty lines from data.txt and save the cleaned content to a new file cleaned.txt.

Concept: Reading lines, filtering out empty lines, and writing back.

## 1. Mini Project Task: Expense Tracker

### Task Description:

Develop an Expense Tracker where users can:

- ✓ Add expenses (Category, Amount, Date)
- ✓ View all expenses
- ✓ Get the total expenditure
- ✓ Store expenses in a file (expenses.txt)

### Key Concepts Used:

- Writing and appending data to a file ("w", "a")
- Reading and processing data ("r")
- String manipulation and calculations

## 2. Mini Project Task: Quiz Score Manager (CSV File)

### Task Description:

Create a Quiz Score Manager that allows users to:

- ✓ Enter quiz scores (Student Name, Subject, Score)
- ✓ Save scores in a CSV file (scores.csv)
- ✓ View all stored scores

✓ Search for a student's score by name

#### Key Concepts Used:

- Handling CSV files using the csv module
- Reading and writing structured data
- Searching within a file

## Day 36

### Exception Handling in Python

#### What is Exception Handling?

Exception handling in Python is a way to manage errors gracefully without stopping program execution. It helps prevent crashes by catching and handling runtime errors.

#### Syntax of Exception Handling in Python

Python provides the try, except, else, and finally blocks to handle exceptions.

try:

```
# Code that may raise an exception
num = int(input("Enter a number: "))
result = 10 / num # May raise ZeroDivisionError
print("Result:", result)
```

except ZeroDivisionError:

```
    print("Error: Cannot divide by zero.")
```

except ValueError:

```

print("Error: Invalid input. Please enter a number.")

else:
    print("No exceptions occurred!")

finally:
    print("Execution completed!") # Always runs

```

## Explanation of Exception Handling Blocks

### 1. try Block

- Contains the code that might raise an error.

### 2. except Block

- Catches and handles the error if it occurs.

### 3. else Block (Optional)

- Runs only if no exception occurs inside the try block.

### 4. finally Block (Optional)

- Always executes, whether an exception occurs or not (useful for cleanup).

## Common Exceptions in Python

Exception	Description
ZeroDivisionError	Division by zero is not allowed.
ValueError	Invalid data type, e.g., entering text instead of a number.
TypeError	Mismatch of data types in operations.
IndexError	Accessing an index that doesn't exist in a list.
KeyError	Accessing a non-existent key in a dictionary.
FileNotFoundException	Trying to open a file that doesn't exist.

## Easy Examples of Exception Handling

### 1. Handling Division by Zero

```
try:
    x = int(input("Enter a number: "))
    print(10 / x) # Error if x is 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

### 2. Handling Invalid User Input

```
try:
    age = int(input("Enter your age: ")) # Error if input is not a number
    print("Your age is:", age)
except ValueError:
    print("Invalid input! Please enter a number.")
```

### 3. Handling Multiple Exceptions

```
try:
    num_list = [10, 20, 30]
    index = int(input("Enter index: "))
    print("Value:", num_list[index]) # May raise IndexError
except (ValueError, IndexError):
    print("Invalid input! Enter a valid number or index.")
```

## Raising Custom Exceptions

You can also create your own exceptions using `raise`.

```
def check_age(age):
    if age < 18:
        raise ValueError("You must be at least 18 years old.")
```

```

else:
    print("Access granted!")

try:
    check_age(16)
except ValueError as e:
    print("Error:", e)

```

## Summary

- ✓ Exception Handling prevents program crashes.
- ✓ Use try, except, else, and finally blocks.
- ✓ Handle specific errors like ZeroDivisionError, ValueError, etc.
- ✓ Custom Exceptions allow better error messages.

## 1. Mini Project: ATM Transaction System

### Task Description:

Create an ATM simulator where users can:

- ✓ Check balance
- ✓ Withdraw money
- ✓ Deposit money
- ✓ Handle errors like:
  - Invalid input (non-numeric values)
  - Insufficient balance
  - Negative deposit/withdrawal amounts

### Key Exception Handling Concepts Used:

- ValueError (for non-numeric input)

- Custom Exception (for insufficient balance)

**Expected Features:**

- BankAccount class with balance
- Methods for deposit(), withdraw(), and check\_balance()
- Exception handling for invalid transactions

```
class InsufficientBalanceError(Exception):
    """Custom exception for insufficient balance."""
    pass

class BankAccount:
    def __init__(self, balance=0):
        self.balance = balance

    def deposit(self, amount):
        try:
            if amount <= 0:
                raise ValueError("Deposit amount must be positive.")
            self.balance += amount
            print(f"₹{amount} deposited successfully. Current balance: ₹{self.balance}")
        except ValueError as e:
            print("Error:", e)

    def withdraw(self, amount):
        try:
            if amount <= 0:
                raise ValueError("Withdrawal amount must be positive.")
            if amount > self.balance:
                raise InsufficientBalanceError("Insufficient balance for this withdrawal.")
            self.balance -= amount
            print(f"₹{amount} withdrawn successfully. Remaining balance: ₹{self.balance}")
        except (ValueError, InsufficientBalanceError) as e:
            print("Error:", e)

    def check_balance(self):
        print(f"Current balance: ₹{self.balance}")
```

```

print(f"Current balance: ₹{self.balance}")

# Testing the ATM system
account = BankAccount(5000) # Starting balance ₹5000

while True:
    print("\n🏧 ATM Menu: 1. Check Balance | 2. Deposit | 3. Withdraw | 4. Exit")
    try:
        choice = int(input("Enter your choice: "))
        if choice == 1:
            account.check_balance()
        elif choice == 2:
            amount = float(input("Enter deposit amount: ₹"))
            account.deposit(amount)
        elif choice == 3:
            amount = float(input("Enter withdrawal amount: ₹"))
            account.withdraw(amount)
        elif choice == 4:
            print("Thank you for using our ATM! 😊")
            break
        else:
            print("Invalid choice! Please enter a valid option.")
    except ValueError:
        print("Invalid input! Please enter a number.")

```

## Features of This ATM System:

- ✓ Handles ValueError if the user enters non-numeric values
- ✓ Custom Exception (InsufficientBalanceError) to prevent overdraft
- ✓ Keeps the program running until the user exits

## 2. Mini Project: Student Marks Grading System

### Task Description:

Create a program that allows students to enter their marks and:

✓ Calculate grade (A, B, C, D, F)

✓ Handle errors like:

- Negative marks
- Marks above 100
- Non-numeric inputs

### Key Exception Handling Concepts Used:

- ValueError (for invalid number input)
- Custom Exception (for out-of-range marks)

### Expected Features:

- Function calculate\_grade(marks)
- Exception handling for invalid input
- Display student's grade based on marks

```
class InvalidMarksError(Exception):
```

```
    """Custom exception for invalid marks."""
```

```
    pass
```

```
def calculate_grade(marks):
```

```
    """Function to determine grade based on marks."""
```

```
    if marks < 0 or marks > 100:
```

```
        raise InvalidMarksError("Marks should be between 0 and 100.")
```

```
    if marks >= 90:
```

```
        return "A"
```

```
    elif marks >= 75:
```

```
        return "B"
```

```
    elif marks >= 60:
```

```

        return "C"
    elif marks >= 40:
        return "D"
    else:
        return "F"

# Taking input from the user
try:
    marks = float(input("Enter student marks (0-100): "))
    grade = calculate_grade(marks)
    print(f"Student Grade: {grade}")

except ValueError:
    print("Error: Please enter a valid number.")
except InvalidMarksError as e:
    print("Error:", e)

```

### **Features of This Grading System:**

- ✓ Handles ValueError if the user enters non-numeric input
- ✓ Custom Exception (InvalidMarksError) for marks out of range
- ✓ Prints the correct grade based on marks

## **Day 36 Tasks**

### **1. Simple Division Program**

- Write a program that asks the user for two numbers and divides them.
- Handle ZeroDivisionError if the user enters 0 as the denominator.

### **2. Handling Invalid Input (ValueError)**

- Ask the user for their age and ensure they enter a valid number.

- Handle ValueError if the user enters non-numeric input.

### 3. List Index Error Handling

- Create a list with 5 elements and ask the user for an index to access.
- Handle IndexError if they enter an out-of-range index.

### 4. KeyError Handling in Dictionary

- Create a dictionary with 3 student names and their marks.
- Ask the user to enter a student name and return their marks.
- Handle KeyError if the name is not in the dictionary.

### 5. File Handling with Exception Handling

- Ask the user for a filename to read.
- Handle FileNotFoundError if the file does not exist.

### 6. Multiple Exception Handling

- Ask the user for a number and divide 100 by it.
- Handle both ZeroDivisionError and ValueError in a single try block.

### 7. Custom Exception for Negative Numbers

- Write a function that raises a custom exception if a number is negative.

### 8. Even Number Checker

- Ask the user for a number and check if it's even.
- Raise a custom exception if the number is odd.

### 9. ATM Withdrawal System

- Create a program where the user can withdraw money from their balance.
- Handle:

- ValueError for invalid input
- Custom Exception for insufficient balance

## 10. Student Marks Validation System

- Ask the user for student marks (0-100).
- Raise a **custom exception** if the marks are negative or above 100.

## 11. Divide and Save Result in File

- Ask the user for two numbers and divide them.
- Save the result in a file.
- Handle ZeroDivisionError, ValueError, and FileNotFoundError.

## 12. Login Authentication with Exception Handling

- Create a system where the user enters a username and password.
- Raise a custom exception if the username is incorrect or the password is too short.

## 13. Exception Logging System

- Modify any program to log errors into a file (errors.log) instead of printing them.
- Use Python's logging module.

## 1. Mini Project: Online Shopping Cart System

### Task Description:

Develop an online shopping cart system where users can:

- ✓ Add items to the cart
- ✓ Remove items from the cart

✓ Checkout and make payment

✓ Handle errors like:

- Invalid product selection (Product not in store)
- Insufficient stock
- Invalid payment amount

#### **Key Exception Handling Concepts Used:**

- KeyError (for selecting an unavailable product)
- ValueError (for non-numeric input)
- Custom Exception (OutOfStockError) for insufficient stock

## **2. Mini Project: Railway Ticket Booking System**

#### **Task Description:**

Create a railway reservation system where users can:

✓ Book a ticket (enter name, destination, seat type)

✓ Cancel a ticket

✓ View ticket details

✓ Handle errors like:

- Invalid seat selection
- Overbooking (limit on available seats)
- Invalid passenger name input

#### **Key Exception Handling Concepts Used:**

- ValueError (for incorrect input format)
- IndexError (for selecting a non-existent seat)
- Custom Exception (BookingFullError) if seats are sold out

# Day 37

## Iterators in Python

### Definition:

An iterator in Python is an object that allows us to traverse (iterate) through a sequence one element at a time without needing to store all elements in memory.

An iterator must implement two special methods:

- `__iter__()` → Returns the iterator object itself.
- `__next__()` → Returns the next value from the iterator. When no more elements are available, it raises a `StopIteration` exception.

### Syntax & Example 1: Using an Iterator

```
# Creating an iterator from a list
numbers = [1, 2, 3, 4, 5]
iterator = iter(numbers) # Convert list to an iterator

# Using next() to access elements one by one
print(next(iterator)) # Output: 1
print(next(iterator)) # Output: 2
print(next(iterator)) # Output: 3
print(next(iterator)) # Output: 4
print(next(iterator)) # Output: 5
# print(next(iterator)) # Raises StopIteration since no elements are left
```

### Explanation:

- We use `iter(numbers)` to convert the list into an iterator.
- The `next(iterator)` function retrieves the next element each time.
- When elements are exhausted, it raises `StopIteration`.

**Example 2: Creating a Custom Iterator**

Let's create a custom iterator that generates numbers from 1 to 5.

```
class MyNumbers:  
    def __iter__(self):  
        self.num = 1  
        return self  
  
    def __next__(self):  
        if self.num > 5:  
            raise StopIteration # Stop when it reaches 5  
        value = self.num  
        self.num += 1  
        return value  
  
# Creating an object of MyNumbers  
my_iter = MyNumbers()  
iterator = iter(my_iter)  
  
# Using a loop to iterate through the iterator  
for num in iterator:  
    print(num) # Output: 1 2 3 4 5
```

**Explanation:**

1. The `__iter__()` method initializes the iterator.
2. The `__next__()` method returns the next number and increments num.
3. When `num > 5`, it raises `StopIteration`, stopping the iteration.

### **Example 3: Using Iterators with for Loop**

Iterators are automatically used in for loops:

```
numbers = [10, 20, 30, 40]
for num in iter(numbers):
    print(num) # Output: 10 20 30 40
```

The for loop internally calls `__next__()` on the iterator.

### **Summary**

- ✓ Iterators help in efficiently processing sequences element by element.
- ✓ Use `iter(object)` to get an iterator and `next(iterator)` to retrieve elements.
- ✓ Define a custom iterator using `__iter__()` and `__next__()`.

## **Mini Project 1: Custom Pagination System using Iterators**

### **Project Overview:**

Create a custom pagination system using iterators to navigate through a list of items (e.g., product listings, student records).

### **Implementation Steps:**

1. Create a `PaginationIterator` class that supports `next()` and `previous()` functionality.
2. Users can navigate through pages using `next()` and `prev()` methods.
3. Each page displays a fixed number of items.

### **Code Implementation:**

```
class PaginationIterator:
    def __init__(self, items, page_size):
```

```
self.items = items
self.page_size = page_size
self.index = 0

def __iter__(self):
    return self

def __next__(self):
    if self.index >= len(self.items):
        raise StopIteration # End of pages
    start = self.index
    end = min(self.index + self.page_size, len(self.items))
    self.index = end # Move to next page
    return self.items[start:end]

def prev(self):
    self.index = max(0, self.index - 2 * self.page_size) # Go back one page
    return next(self)

# Sample data
products = ["Laptop", "Mouse", "Keyboard", "Monitor", "Printer", "Webcam",
            "Headset", "Speaker"]
page_size = 3

# Using the iterator
pager = PaginationIterator(products, page_size)
print(next(pager)) # Output: ['Laptop', 'Mouse', 'Keyboard']
print(next(pager)) # Output: ['Monitor', 'Printer', 'Webcam']
print(pager.prev()) # Output: ['Laptop', 'Mouse', 'Keyboard']
```

**Features:**

- ✓ Supports pagination navigation.
- ✓ Can go forward and backward.
- ✓ Uses iterators to efficiently load data.

**Mini Project 2: Custom File Reader Iterator****Project Overview:**

Create an iterator-based file reader that reads N lines at a time, useful for handling large files efficiently.

**Implementation Steps:**

1. Create a FileReaderIterator class that reads a file in chunks of N lines.
2. Implement `__iter__()` and `__next__()` methods.
3. Use lazy loading to avoid loading the entire file into memory.

**Code Implementation:**

```
class FileReaderIterator:
    def __init__(self, filename, chunk_size=3):
        self.file = open(filename, "r")
        self.chunk_size = chunk_size
        self.lines = []

    def __iter__(self):
        return self

    def __next__(self):
        self.lines = [self.file.readline().strip() for _ in range(self.chunk_size)]
        if not any(self.lines): # Stop if no more lines
            raise StopIteration
        return self.lines
```

```
self.file.close()
raise StopIteration
return self.lines

# Create a sample file (for testing)
with open("sample.txt", "w") as file:
    file.write("Line 1\nLine 2\nLine 3\nLine 4\nLine 5\nLine 6\nLine 7\nLine 8\n")

# Using the iterator
reader = FileReaderIterator("sample.txt", chunk_size=2)
print(next(reader)) # Output: ['Line 1', 'Line 2']
print(next(reader)) # Output: ['Line 3', 'Line 4']
print(next(reader)) # Output: ['Line 5', 'Line 6']
print(next(reader)) # Output: ['Line 7', 'Line 8']
```

### Features:

- ✓ Reads files in chunks, preventing memory overload.
- ✓ Lazy loading approach ensures efficiency.
- ✓ Useful for large text files.

### Summary

1. Custom Pagination Iterator for navigating through lists.
2. File Reader Iterator for reading files in chunks.

### Day 37 Tasks

1. Create a Custom Iterator: Write a class that implements `__iter__()` and `__next__()` to iterate over numbers from 1 to 10.

2. Iterate Over a List Using an Iterator: Use the `iter()` and `next()` functions to manually iterate over a given list of colors.
3. Create a Reverse Iterator: Implement a custom iterator that iterates over a string in reverse order.
4. Iterator for Even Numbers: Create an iterator that returns only even numbers from a given range (1 to 20).
5. Manually Iterate Over a Tuple: Given a tuple of names, use `iter()` and `next()` to iterate manually.
6. Implement a Countdown Iterator: Write an iterator that starts from a given number (e.g., 10) and counts down to 1.
7. Custom Step Iterator: Build an iterator that iterates over a range but allows custom steps (e.g., step size of 3: 0, 3, 6, 9...).
8. Circular Iterator: Implement an iterator that repeats elements indefinitely (e.g., cycling through ["A", "B", "C"]).
9. Prime Number Iterator: Create an iterator that generates prime numbers up to a given limit (e.g., 50).
10. Fibonacci Sequence Iterator: Implement an iterator that generates Fibonacci numbers up to a given limit.
11. Iterate Over a File Line by Line: Open a text file and read lines one by one using an iterator instead of loading the full file into memory.
12. Custom Data Stream Iterator: Create an iterator that reads live data (e.g., stock prices or sensor data) in real-time using `yield`.
13. Pagination Iterator for Large Data Processing: Implement a pagination system using an iterator where each page returns 5 items from a large dataset.

## Summary

- ✓ 6 Beginner tasks for understanding basic iterators.
- ✓ 4 Intermediate tasks for real-world applications.
- ✓ 3 Advanced tasks for handling large data efficiently.

## Mini Project Task 1: Custom Playlist Iterator

### Project Overview:

Create a music playlist iterator that allows users to:

- ✓ Play songs one by one.
- ✓ Go to the next or previous song.
- ✓ Loop back to the start when reaching the end.

### Implementation Steps:

1. Create a PlaylistIterator class that stores a list of songs.
2. Implement `__iter__()` and `__next__()` to play songs in order.
3. Add a method to move back to the previous song.
4. If the last song is reached, loop back to the first song.

## Mini Project Task 2: Batch Data Processor

### Project Overview:

Build an iterator-based data processor that:

- ✓ Reads and processes data in batches (e.g., student records, log files).
- ✓ Handles large datasets efficiently without loading everything into memory.
- ✓ Uses iterators to fetch N records at a time.

### Implementation Steps:

1. Create a BatchDataIterator class to read N records at a time.
2. Implement `__iter__()` and `__next__()` for batch processing.
3. If no more data is available, stop iteration gracefully.

# Day 38

## Generators in Python

### Definition:

A generator in Python is a special type of iterator that allows lazy evaluation, meaning it generates values on the fly instead of storing them in memory.

- Generators are defined using functions with the `yield` keyword instead of `return`.
- They improve performance, especially when handling large datasets.

### Syntax of Generators

```
def generator_function():
    yield "Hello"
    yield "World"

gen = generator_function()
print(next(gen)) # Output: Hello
print(next(gen)) # Output: World
```

### Explanation:

- The function does not return values immediately; instead, it yields them one at a time.
- The state of the function is saved after each `yield`, so it resumes from where it left off.

## Example 1: Simple Generator for Counting Numbers

```
def count_up_to(n):
    count = 1
    while count <= n:
        yield count
        count += 1 # The function resumes here in the next call

# Creating generator
counter = count_up_to(5)

# Using next() to retrieve values
print(next(counter)) # Output: 1
print(next(counter)) # Output: 2
print(next(counter)) # Output: 3
print(next(counter)) # Output: 4
print(next(counter)) # Output: 5
# print(next(counter)) # Raises StopIteration as the generator is exhausted
```

### Explanation:

- The function yields values one by one up to n.
- The state is preserved, so when next() is called again, execution resumes from the last yield.

## Example 2: Generator for Fibonacci Series

```
def fibonacci(limit):
    a, b = 0, 1
    while a < limit:
        yield a
        a, b = b, a + b # Move to the next number
# Using the generator
```

```

fib_gen = fibonacci(10)
for num in fib_gen:
    print(num) # Output: 0 1 1 2 3 5 8

```

**Explanation:**

- This generator dynamically generates Fibonacci numbers up to a given limit.
- It does not store the whole sequence in memory.

**Example 3: Generator Expression (Shorter Syntax)**

Like list comprehensions, Python has generator expressions:

```
gen_exp = (x * x for x in range(5))
```

```

print(next(gen_exp)) # Output: 0
print(next(gen_exp)) # Output: 1
print(next(gen_exp)) # Output: 4
print(next(gen_exp)) # Output: 9
print(next(gen_exp)) # Output: 16

```

**Key Differences: Generator vs Iterator**

Feature	Generators	Iterators
Creation	Defined using yield	Implements <code>__iter__()</code> and <code>__next__()</code>
Memory Usage	Efficient, generates values on demand	Can consume more memory
State Saving	Saves state automatically	Requires manual state handling
Complexity	Easier to implement	More complex

## Summary

- ✓ Generators are memory-efficient and process data lazily.
- ✓ Use yield instead of return to pause and resume execution.
- ✓ Can be used with next() or for loops for iteration.
- ✓ Generator expressions provide a concise way to create generators.

## Mini Project 1: Infinite Fibonacci Generator

### Project Overview:

Create a generator function that produces an infinite sequence of Fibonacci numbers.

- ✓ Efficiently generates Fibonacci numbers using yield.
- ✓ Avoids storing large lists in memory.
- ✓ Users can request as many numbers as they need.

### Implementation Steps:

1. Define a function fibonacci\_generator() that yields Fibonacci numbers.
2. Use a while True loop to generate numbers infinitely.
3. Call next() to get numbers one by one.

### Code Implementation:

```
def fibonacci_generator():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b # Move to the next Fibonacci number
```

```
# Using the generator
fib_gen = fibonacci_generator()

# Generate the first 10 Fibonacci numbers
for _ in range(10):
    print(next(fib_gen))
```

#### Sample Output:

```
0
1
1
2
3
5
8
13
21
34
```

## Mini Project 2: Log File Reader

#### Project Overview:

Build a log file reader using generators that:

- ✓ Reads large log files line by line without loading everything into memory.
- ✓ Can process files efficiently for error tracking or monitoring.
- ✓ Returns lines containing specific keywords like "ERROR".

#### Implementation Steps:

1. Define a function `log_reader()` that yields lines one by one.
2. Use `yield` to read large files efficiently.

3. Allow filtering by keywords (e.g., "ERROR", "WARNING").

**Code Implementation:**

```
def log_reader(filename, keyword):
    with open(filename, "r") as file:
        for line in file:
            if keyword in line:
                yield line.strip()

# Using the generator
log_gen = log_reader("server_logs.txt", "ERROR")

# Print all error messages from the log file
for error_line in log_gen:
    print(error_line)
```

**Sample Log File (server\_logs.txt):**

INFO: Server started successfully  
 ERROR: Database connection failed  
 WARNING: High memory usage detected  
 ERROR: User authentication failed  
 INFO: Request processed successfully

**Sample Output (Filtering "ERROR")**

ERROR: Database connection failed  
 ERROR: User authentication failed

**Summary**

1. Infinite Fibonacci Generator  – Generates Fibonacci numbers dynamically.
2. Log File Reader  – Reads large files line by line with keyword filtering.

## Day 38 Tasks

1. Create a simple generator that yields numbers from 1 to 10.
2. Write a generator for even numbers between 1 and 50.
3. Create a generator that yields squares of numbers from 1 to 10.
4. Build a generator to produce the first N prime numbers.
5. Write a generator function that yields characters of a given string one by one.
6. Implement a Fibonacci sequence generator that generates infinite Fibonacci numbers.
7. Create a countdown generator that counts down from a given number to zero.
8. Develop a generator that reads lines from a text file one by one without loading the entire file into memory.
9. Create a generator for generating random passwords with uppercase, lowercase, numbers, and special characters.
10. Build a generator that continuously yields alternating positive and negative numbers (e.g., 1, -1, 2, -2, 3, -3...).
11. Implement a log file reader generator that filters and returns only error messages from a large log file.
12. Write a generator that yields numbers following a mathematical pattern, such as the triangular number sequence.
13. Create a generator that produces unique session IDs for users in a web application.

## Mini Project 1: CSV File Data Streamer

### Project Overview:

Create a generator function that reads a large CSV file line by line without loading it entirely into memory.

- ✓ Efficiently reads and processes large CSV files.
- ✓ Handles millions of records without memory issues.
- ✓ Allows filtering specific rows based on a condition (e.g., salary > 50,000).

### Implementation Steps:

1. Define a `csv_reader()` generator function that reads a CSV file line by line.
2. Use Python's built-in `csv` module for parsing the file.
3. Allow filtering by specific conditions (e.g., `Salary > 50000`).
4. Use `yield` to return rows one at a time.

### Sample CSV (`employees.csv`)

```
Name,Salary,Department
Alice,60000,IT
Bob,45000,HR
Charlie,75000,Finance
David,40000,Marketing
Emma,90000,IT
```

### Expected Output

```
{"Name": "Alice", "Salary": "60000", "Department": "IT"}
{"Name": "Charlie", "Salary": "75000", "Department": "Finance"}
{"Name": "Emma", "Salary": "90000", "Department": "IT"}
```

## Mini Project 2: Paginated API Data Fetcher

### Project Overview:

Create a generator function that fetches data from an API in pages to avoid overloading memory.

- ✓ Fetches data one page at a time.
- ✓ Works with paginated APIs like GitHub, OpenWeather, or Twitter.
- ✓ Useful for handling large API responses efficiently.

### Implementation Steps:

1. Use Python's requests module to fetch data page by page.
2. Define a generator function `api_data_fetcher()` that yields one page at a time.
3. Automatically stop when no more pages are available.
4. Print results one page at a time instead of loading everything at once.

### Summary

1. CSV File Data Streamer  – Reads large CSV files efficiently.
2. Paginated API Data Fetcher  – Fetches large API responses page by page.

# Day 39

## Decorators in Python

### Definition:

A decorator in Python is a function that modifies the behavior of another function without changing its structure. It allows us to add functionality to functions dynamically.

- Uses the @decorator\_name syntax.
- Commonly used for logging, authentication, and timing functions.

### Syntax of Decorators

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper
```

```
@my_decorator # Applying the decorator
def say_hello():
    print("Hello, World!")

say_hello()
```

### Output:

Something is happening before the function is called.  
Hello, World!  
Something is happening after the function is called.

**Explanation:**

- The wrapper() function modifies the behavior of say\_hello().
- The @my\_decorator syntax automatically passes say\_hello() to my\_decorator.
- The function say\_hello() is executed within the wrapper().

**Example 1: Logging with Decorators**

```
def log_decorator(func):
    def wrapper():
        print(f"Calling function {func.__name__}")
        func()
        print(f"Function {func.__name__} finished execution")
    return wrapper

@log_decorator
def greet():
    print("Hello, Python!")

greet()
```

**Output:**

```
Calling function greet
Hello, Python!
Function greet finished execution
```

**Example 2: Decorator with Arguments (Using \*args and \*\*kwargs)**

```
def smart_divide(func):
    def wrapper(a, b):
        if b == 0:
            print("Cannot divide by zero!")
            return
        return func(a, b)
    return wrapper

@smart_divide
def divide(a, b):
    return a / b

print(divide(10, 2)) # Output: 5.0
print(divide(10, 0)) # Output: Cannot divide by zero!
```

**Explanation:**

- wrapper() checks if b == 0 before calling the function.
- If b is 0, it prevents division by zero.

**Example 3: Measuring Execution Time (Performance Monitoring)**

```
import time
```

```
def timer_decorator(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"{func.__name__} took {end_time - start_time:.4f} seconds")
        return result
    return wrapper
```

```
@timer_decorator
def slow_function():
    time.sleep(2)
    print("Function finished!")
slow_function()
```

**Output:**

```
Function finished!
slow_function took 2.0001 seconds
```

**Example 4: Applying Multiple Decorators**

```
def uppercase_decorator(func):
    def wrapper():
        result = func()
        return result.upper()
    return wrapper
```

```
def exclamation_decorator(func):
    def wrapper():
        result = func()
        return result + "!!!"
    return wrapper
```

```
@exclamation_decorator
@uppercase_decorator
def say_message():
    return "hello"
```

```
print(say_message()) # Output: HELLO!!!
```

### Explanation:

- First, uppercase\_decorator converts text to uppercase.
- Then, exclamation\_decorator adds "!!!" at the end.

### Key Takeaways

- ✓ Decorators allow adding functionality without modifying function code.
- ✓ Used for logging, performance monitoring, authentication, and validation.
- ✓ Can handle functions with arguments using \*args and \*\*kwargs.
- ✓ Multiple decorators can be applied to a function.

## Mini Project 1: Timing Function Decorator ↗

### Project Overview:

Create a decorator function to measure the execution time of any function.

- ✓ Useful for performance testing or optimization.
- ✓ Can be used to log how long different functions take to execute.

### Implementation Steps:

1. Define a decorator timer\_decorator() that wraps any function.
2. Use time.time() to calculate the start and end time of the function.
3. Print or log the elapsed time for each function execution.

### Code Implementation:

```
import time
```

```
# Decorator function
def timer_decorator(func):
    def wrapper(*args, **kwargs):
        start_time = time.time() # Start time
        result = func(*args, **kwargs) # Call the actual function
        end_time = time.time() # End time
        print(f"{func.__name__} executed in {end_time - start_time:.4f} seconds")
        return result
    return wrapper
```

```
# Example function to demonstrate the decorator
@timer_decorator
def long_running_function():
    time.sleep(2) # Simulate a time-consuming task
    return "Task Completed"
```

```
# Call the decorated function
print(long_running_function())
```

#### **Sample Output:**

```
long_running_function executed in 2.0001 seconds
Task Completed
```

## **Mini Project 2: Access Control Decorator**

#### **Project Overview:**

Create a decorator that controls access to certain functions based on a user's role (e.g., admin, guest).

- ✓ Can be used to secure sensitive functions or restrict access to certain functionalities.
- ✓ Allows for easy role-based access control across different functions.

### **Implementation Steps:**

1. Define a decorator role\_required() that checks the user's role.
2. Pass the role as an argument and decide whether the user can access the function.
3. If the role is invalid or unauthorized, raise an exception or return an error message.

### **Code Implementation:**

```
# Decorator for access control based on role
def role_required(role):
    def decorator(func):
        def wrapper(user_role, *args, **kwargs):
            if user_role != role:
                raise PermissionError(f"Access Denied: {role} role required")
            return func(user_role, *args, **kwargs)
        return wrapper
    return decorator

# Example function to demonstrate the decorator
@role_required("admin")
def delete_user(user_role, username):
    return f"User {username} has been deleted."

# Test with different roles
try:
    print(delete_user("admin", "john_doe")) # Allowed
    print(delete_user("guest", "john_doe")) # Denied
except PermissionError as e:
    print(e)
```

### Sample Output:

User john\_doe has been deleted.  
Access Denied: admin role required

### Summary

1. Timing Function Decorator  – Measures the execution time of any function.
2. Access Control Decorator  – Restricts access based on user roles.

### Day 39 Tasks

1. Create a basic generator that yields numbers from 1 to 5.
2. Write a generator that returns the square of each number from 1 to 10.
3. Implement a generator that generates Fibonacci numbers up to a given number.
4. Create a generator that yields odd numbers between 1 and 50.
5. Write a generator that takes a list of strings and yields each string's length.
6. Create a generator that reads a file line by line, yielding each line (for large files).
7. Write a generator that generates prime numbers up to a given limit.
8. Build a generator that mimics the behavior of range() by yielding numbers up to a given limit with a specific step.
9. Create a generator that converts a list of words into uppercase one at a time.
10. Write a generator that fetches data from an API and yields the results page by page (for paginated APIs).
11. Create a generator for an infinite sequence of numbers that increments by 1.
12. Write a generator that continuously alternates between two values (e.g., "A", "B", "A", "B"...).
13. Build a generator that simulates the rolling of a dice, yielding random values between 1 and 6.

## Mini Project 1: Lazy Loading Image Viewer

### Project Overview:

Create a generator that **lazily loads images** from a folder, displaying one image at a time. This is useful when working with a large number of images, avoiding loading all images into memory at once.

### Implementation Steps:

1. Create a folder with multiple image files (e.g., .jpg, .png).
2. Write a generator `lazy_load_images()` that reads the folder and yields one image at a time.
3. Display each image sequentially, pausing after each one until the user decides to continue.

### Expected Outcome:

- Images are loaded and displayed one by one.
- The user presses Enter to proceed to the next image, without overloading memory.

## Mini Project 2: Log File Parser

### Project Overview:

Write a generator that reads a log file line by line and filters out specific types of log entries (e.g., errors or warnings). This is useful for parsing large log files efficiently.

### Implementation Steps:

1. Prepare a large log file with various log entries (e.g., info, warning, error).
2. Create a generator `filter_log_entries()` that reads the file and yields only the lines that contain "ERROR" or "WARNING".

3. Display or process the filtered log entries.

**Expected Outcome:**

- The program will filter and display log entries with the specified type (e.g., "ERROR").
- The generator will only load and process relevant log entries, improving memory efficiency when dealing with large log files.

## NumPy

### Day 40

#### Introduction to NumPy & Basics

##### What is NumPy?

NumPy (Numerical Python) is a powerful library in Python used for numerical computations. It provides support for large, multi-dimensional arrays and matrices, along with mathematical functions to operate on these structures efficiently.

##### Importance of NumPy in Data Science

- Efficient Computation: Faster than Python lists due to optimized C implementation.
- Multidimensional Support: Handles large datasets efficiently.
- Mathematical Operations: Supports advanced operations like linear algebra, statistics, and random number generation.
- Integration: Works well with Pandas, Matplotlib, and SciPy.

## Installing NumPy

You can install NumPy using pip:

```
pip install numpy
```

## Importing NumPy

To use NumPy in Python, import it using:

```
import numpy as np
```

Here, np is an alias for NumPy, making it easier to use.

## Creating NumPy Arrays

NumPy arrays are more efficient than Python lists for numerical operations.

### Using np.array()

The np.array() function is used to create a NumPy array.

#### Syntax:

```
np.array(object, dtype=None)
```

- object: The input list or tuple.
- dtype: Data type of the array (optional).

#### Example:

```
import numpy as np
```

```
arr = np.array([1, 2, 3, 4, 5])
print(arr) # Output: [1 2 3 4 5]
```

## Creating 1D, 2D, and 3D Arrays

### 1. One-Dimensional (1D) Array

```
arr1D = np.array([10, 20, 30, 40])  
print(arr1D)
```

**Output:** [10 20 30 40]

### 2. Two-Dimensional (2D) Array (Matrix)

```
arr2D = np.array([[1, 2, 3], [4, 5, 6]])  
print(arr2D)
```

**Output:**

```
[[1 2 3]  
 [4 5 6]]
```

### 3. Three-Dimensional (3D) Array (Tensor)

```
arr3D = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])  
print(arr3D)
```

**Output:**

```
[[[1 2]  
 [3 4]]
```

```
[[[5 6]  
 [7 8]]]
```

## Checking NumPy Version

To check the installed NumPy version:

```
print(np.__version__)
```

**Example Output:** 2.1.3

## NumPy Performance vs Python Lists

NumPy is much faster than Python lists because:

- ✓ Uses fixed-size memory storage.
- ✓ Performs vectorized operations (instead of loops).
- ✓ Written in optimized C language.

### Example: Comparing Execution Time

```
import numpy as np
```

```
import time
```

```
# Using Python List
```

```
lst = list(range(1000000))
start = time.time()
lst = [x*2 for x in lst]
end = time.time()
print("Python List Time:", end - start)
```

```
# Using NumPy Array
```

```
arr = np.arange(1000000)
start = time.time()
arr = arr * 2
end = time.time()
print("NumPy Array Time:", end - start)
```

NumPy is significantly faster!

## Basic Array Attributes

NumPy arrays have several useful properties:

Attribute	Description
.ndim	Number of dimensions (axes)
.shape	Shape (size of each dimension)
.size	Total number of elements
.dtype	Data type of elements
.itemsize	Size of each element in bytes

**Example:**

```
arr = np.array([[1, 2, 3], [4, 5, 6]])

print(arr.ndim)    # Output: 2 (2D array)
print(arr.shape)   # Output: (2, 3) → 2 rows, 3 columns
print(arr.size)    # Output: 6 (total elements)
print(arr.dtype)   # Output: int64 (data type)
print(arr.itemsize) # Output: 4 (bytes per element)
```

## Indexing & Slicing in 1D Arrays

NumPy supports indexing (accessing elements) and slicing (extracting subarrays).

### Indexing (Accessing Elements)

```
arr = np.array([10, 20, 30, 40, 50])
print(arr[0]) # Output: 10
print(arr[2]) # Output: 30
print(arr[-1]) # Output: 50 (Last element)
```

### Slicing (Extracting Subarrays)

Syntax: array[start:end:step]

```
arr = np.array([10, 20, 30, 40, 50, 60, 70])

print(arr[1:5]) # Output: [20 30 40 50]
```

```
print(arr[:4]) # Output: [10 20 30 40] (First 4 elements)
print(arr[2:]) # Output: [30 40 50 60 70] (From index 2 to end)
print(arr[::-2]) # Output: [10 30 50 70] (Every 2nd element)
print(arr[::-1]) # Output: [70 60 50 40 30 20 10] (Reversed)
```

## Summary

- ✓ NumPy is a fast and powerful library for numerical computing.
- ✓ NumPy arrays are more efficient than Python lists.
- ✓ Supports 1D, 2D, and 3D arrays for complex data.
- ✓ Attributes like .ndim, .shape, .size, etc., help analyze arrays.
- ✓ Indexing & Slicing allows flexible data selection.

## Mini Project 1: Student Marks Analyzer

### Objective:

Create a Python program that takes a list of student marks, converts them into a NumPy array, and calculates useful statistics like the average, highest, lowest, and total marks.

### Features Covered:

- ✓ Installing and Importing NumPy
- ✓ Creating NumPy Arrays (1D)
- ✓ Checking NumPy Version
- ✓ Using .ndim, .shape, .size, .dtype
- ✓ Indexing & Slicing

**Code Implementation:**

```
# Step 1: Import NumPy
```

```
import numpy as np
```

```
# Step 2: Check NumPy Version
```

```
print("❖ NumPy Version:", np.__version__)
```

```
# Step 3: Take student marks as input and create a NumPy array
```

```
marks = input("Enter student marks separated by spaces: ").split()
```

```
marks_array = np.array(marks, dtype=int) # Convert input to integer NumPy array
```

```
# Step 4: Display the created array
```

```
print("\n❖ Student Marks Array:", marks_array)
```

```
# Step 5: Display array attributes
```

```
print("\n❖ Array Attributes:")
```

```
print("❖ Dimensions:", marks_array.ndim)
```

```
print("❖ Shape:", marks_array.shape)
```

```
print("❖ Size:", marks_array.size)
```

```
print("❖ Data Type:", marks_array.dtype)
```

```
# Step 6: Perform basic analysis
```

```
print("\n❖ Marks Analysis:")
```

```
print("❖ Highest Mark:", np.max(marks_array))
```

```
print("❖ Lowest Mark:", np.min(marks_array))
```

```
print("❖ Average Mark:", np.mean(marks_array))
```

```
print("❖ Total Marks:", np.sum(marks_array))
```

```
# Step 7: Slicing operations
```

```
print("\nQ Slicing Examples:")
print("First 3 students' marks:", marks_array[:3])
print("Last 2 students' marks:", marks_array[-2:])
print("Marks of alternate students:", marks_array[::2])
```

#### **Sample Output:**

NumPy Version: 1.21.2

Enter student marks separated by spaces: 78 85 90 66 92 88

Student Marks Array: [78 85 90 66 92 88]

#### **■ Array Attributes:**

- ◆ Dimensions: 1
- ◆ Shape: (6,)
- ◆ Size: 6
- ◆ Data Type: int32

#### **■ Marks Analysis:**

🏆 Highest Mark: 92

⚡ Lowest Mark: 66

■ Average Mark: 83.16

■ Total Marks: 499

#### **Slicing Examples:**

First 3 students' marks: [78 85 90]

Last 2 students' marks: [92 88]

Marks of alternate students: [78 90 92]

## What You Learned?

- Used NumPy arrays to store student marks.
- Performed basic statistical calculations (max, min, mean, sum).
- Explored indexing and slicing in 1D arrays.

## Mini Project 2: 2D NumPy Matrix Operations

### Objective:

Create and manipulate a 2D NumPy array (matrix) representing employee salaries, then calculate row-wise and column-wise statistics.

### Features Covered:

- ✓ Creating a 2D NumPy array
- ✓ Understanding .ndim, .shape, .size, .dtype
- ✓ Matrix Indexing & Slicing
- ✓ Row-wise and Column-wise Operations

### Code Implementation:

```
# Step 1: Import NumPy
import numpy as np

# Step 2: Create a 2D NumPy array representing employee salaries
salaries = np.array([
    [45000, 52000, 60000], # Employee 1
    [47000, 55000, 62000], # Employee 2
    [50000, 53000, 65000], # Employee 3
])
```

```

# Step 3: Display the 2D NumPy array
print("↙ Employee Salaries Matrix:\n", salaries)

# Step 4: Display array attributes
print("\n☰ Array Attributes:")
print("◆ Dimensions:", salaries.ndim)
print("◆ Shape:", salaries.shape)
print("◆ Size:", salaries.size)
print("◆ Data Type:", salaries.dtype)

# Step 5: Row-wise and Column-wise Operations
print("\n☛ Salary Analysis:")
print("🏆 Highest Salary in each row:", np.max(salaries, axis=1))
print("⚡ Lowest Salary in each column:", np.min(salaries, axis=0))
print("〽️ Average Salary per Employee:", np.mean(salaries, axis=1))
print("〽️ Total Salary Expense per Month:", np.sum(salaries))

# Step 6: Slicing operations
print("\n🔍 Slicing Examples:")
print("First two employees' salaries:\n", salaries[:2])
print("Salaries of last column (last month):", salaries[:, -1])

```

**Sample Output:**

Employee Salaries Matrix:  
[[45000 52000 60000]  
[47000 55000 62000]  
[50000 53000 65000]]

〽️ Array Attributes:

◆ Dimensions: 2

◆ Shape: (3, 3)

◆ Size: 9

◆ Data Type: int32

### ☛ Salary Analysis:

🏆 Highest Salary in each row: [60000 62000 65000]

🏆 Lowest Salary in each column: [45000 52000 60000]

🏆 Average Salary per Employee: [52333.33 54666.67 56000.00]

🏆 Total Salary Expense per Month: 429000

### Slicing Examples:

First two employees' salaries:

[[45000 52000 60000]

[47000 55000 62000]]

Salaries of last column (last month): [60000 62000 65000]

### What You Learned?

- Created a 2D NumPy array (matrix) for employee salaries.
- Used row-wise and column-wise operations with axis=0 and axis=1.
- Explored matrix indexing and slicing techniques.

### Summary

Mini Project	Concepts Covered
Student Marks Analyzer	1D NumPy array, slicing, basic stats
2D NumPy Matrix Operations	2D NumPy array, matrix indexing, row/column operations

## Day 40 Tasks

1. **Installation:** How do you install NumPy in your Python environment? Provide the command.
2. installed version.
3. **1D Array Creation:** Create a 1D NumPy array with five elements and print it.
4. **2D Array Creation:** Create a 2D NumPy array (2x3 matrix) and print it.
5. **3D Array Creation:** Create a 3D NumPy array and print it.
6. **Array Attributes:** Write a program to display the following properties of a NumPy array:

Number of dimensions (ndim)

Shape (shape)

Size (size)

Data type (dtype)

Memory size of each element (itemsize)

7. **Float Array:** Create a NumPy array with three elements of float type and print it.

8. **Convert List to NumPy Array:** Convert a Python list [5, 10, 15, 20] into a NumPy array and print both.

9. **Performance Test:** Compare the execution time of multiplying a large list and a NumPy array by 2.

10. **Indexing in 1D Array:** Given an array [10, 20, 30, 40, 50], write a program to access:

The first element

The last element

The third element

11. Slicing in 1D Array: Given an array [5, 10, 15, 20, 25, 30], extract and print:

The first three elements

The last two elements

Every alternate element

12. Zeros & Ones Array: Create a 3x3 matrix of zeros and a 2x2 matrix of ones using NumPy.

13. (2x3) and print it.

## Mini Projects

### 1. Project: NumPy Data Organizer

**Task:** Create a program that organizes and reshapes data using NumPy arrays.

#### Steps to Follow:

1. Install and import NumPy (pip install numpy and import numpy as np).
2. Create a 1D NumPy array with 12 random numbers between 1 and 100.
3. Reshape the 1D array into a 2D array (4x3 matrix).
4. Extract and print:
  - a. The second row of the matrix.
  - b. The first column of the matrix.
  - c. The last two elements from the last row.
5. Print the array's shape, size, and data type.

#### Expected Outcome:

- Display the reshaped matrix and extracted elements.
- Print array properties like shape, size, and dtype.

## 2. Project: NumPy Statistics Calculator

**Task:** Build a NumPy-based statistics calculator for numerical datasets.

**Steps to Follow:**

1. Install and import NumPy.
2. Create a 1D NumPy array with at least 10 numerical values.
3. Calculate and display:
  - a. Mean
  - b. Median
  - c. Standard Deviation
  - d. Minimum and Maximum values
4. Slice the first five elements and calculate their average.

**Expected Outcome:**

- Display statistical measures of the dataset.
- Demonstrate slicing and calculations on selected values.

## Day 41

### 1. Array Creation

In NumPy, there are various ways to create arrays, either initialized with specific values or filled with random numbers.

#### a) np.zeros()

- **Definition:** Creates a NumPy array of the specified shape and fills it with zeros.
- **Syntax:**

`np.zeros(shape, dtype=float)`

- **Parameters:**

- shape: Shape of the array (can be a tuple like (3, 2) for a 3x2 array).
- dtype: Data type of the array elements (default is float).

- **Example:**

```
import numpy as np  
arr = np.zeros((3, 3))  
print(arr)
```

- **Output:**

```
[[0. 0. 0.]  
 [0. 0. 0.]  
 [0. 0. 0.]]
```

### b) **np.ones()**

- **Definition:** Creates a NumPy array of the specified shape and fills it with ones.
- **Syntax:**

```
np.ones(shape, dtype=float)
```

- **Parameters:** Same as np.zeros().
- **Example:**

```
arr = np.ones((2, 4), dtype=int)  
print(arr)
```

- **Output:**

```
[[1 1 1 1]  
 [1 1 1 1]]
```

### c) np.full()

- **Definition:** Creates an array filled with a specific value.
- **Syntax:**

```
np.full(shape, fill_value, dtype=None)
```

- **Parameters:**
  - shape: Shape of the array.
  - fill\_value: The value to fill the array with.
  - dtype: Data type of the array elements (optional).
- **Example:**

```
arr = np.full((2, 3), 7)
print(arr)
```

- **Output:**

```
[[7 7 7]
 [7 7 7]]
```

### d) np.eye()

- **Definition:** Creates a 2D array with ones on the diagonal and zeros elsewhere (identity matrix).
- **Syntax:**

```
np.eye(N, M=None, dtype=float)
```

- **Parameters:**
  - N: Number of rows.
  - M: Number of columns (if not provided, it defaults to N).
  - dtype: Data type (optional).
- **Example:**

```
arr = np.eye(3)
```

```
print(arr)
```

- **Output:**

```
[[1. 0. 0.]
```

```
[0. 1. 0.]
```

```
[0. 0. 1.]]
```

### e) np.arange()

- **Definition:** Creates an array with regularly spaced values (like range() in Python).

- **Syntax:**

```
np.arange([start,] stop[, step])
```

- **Parameters:**

- start: Start of the range (default is 0).
- stop: End of the range.
- step: Step between values (default is 1).

- **Example:**

```
arr = np.arange(5, 20, 3)
```

```
print(arr)
```

- **Output:**

```
[ 5  8 11 14 17]
```

### f) np.linspace()

- **Definition:** Creates an array with a specified number of equally spaced values over a given interval.

- **Syntax:**

```
np.linspace(start, stop, num=50)
```

- **Parameters:**

- start: Starting value.
- stop: Ending value.
- num: Number of equally spaced samples to generate (default is 50).

- **Example:**

```
arr = np.linspace(0, 10, 5)
print(arr)
```

- **Output:**

[ 0. 2.5 5. 7.5 10. ]

## g) Random Arrays

1. **np.random.rand()**: Returns an array of the given shape filled with random values between 0 and 1.

- a. **Syntax:**

```
np.random.rand(d0, d1, ..., dn)
```

- b. **Example:**

```
arr = np.random.rand(2, 3)
print(arr)
```

- c. **Output:**

```
[[0.5597344 0.1523804 0.25355145]
 [0.31223593 0.67269397 0.68906356]]
```

2. **np.random.randn()**: Returns an array of the given shape with random values from a **standard normal distribution** (mean = 0, std = 1).

- a. **Syntax:**

```
np.random.randn(d0, d1, ..., dn)
```

- b. **Example:**

```
arr = np.random.randn(2, 3)
print(arr)
```

c. **Output:**

```
[[ 0.28434669 -0.85049483  0.56938498]
 [-0.36066326 -0.54337918  0.36741027]]
```

3. **np.random.randint()**: Returns random integers from a specified range.

a. **Syntax:**

```
np.random.randint(low, high, size)
```

b. **Example:**

```
arr = np.random.randint(1, 10, size=(3, 4))
print(arr)
```

c. **Output:**

```
[[9 5 4 7]
 [1 3 1 2]
 [4 9 8 8]]
```

## 2. Reshaping Arrays

### a) .reshape()

- **Definition:** Reshapes an array to a new shape without changing its data.
- **Syntax:**

```
arr.reshape(new_shape)
```

- **Example:**

```
arr = np.arange(9)
reshaped_arr = arr.reshape((3, 3))
print(reshaped_arr)
```

- **Output:**

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

### b) .ravel()

- **Definition:** Flattens a multi-dimensional array into a 1D array.  
(Unlike reshape(), ravel() always returns a view of the original array).

- **Syntax:**

```
arr.ravel()
```

- **Example:**

```
arr = np.array([[1, 2], [3, 4]])
ravel_arr = arr.ravel()
print(ravel_arr)
```

- **Output:**

```
[1 2 3 4]
```

### c) .flatten()

- **Definition:** Flattens a multi-dimensional array into a 1D array.  
(Unlike ravel(), flatten() always returns a copy).

- **Syntax:**

```
arr.flatten()
```

- **Example:**

```
arr = np.array([[1, 2], [3, 4]])
flattened_arr = arr.flatten()
print(flattened_arr)
```

- **Output:**

[1 2 3 4]

### 3. Copying & Cloning Arrays

#### a) copy() vs view()

- **Definition:**

- `copy()` creates a **new** independent array (a copy of the original).
- `view()` creates a **view** of the original array, meaning changes to the view affect the original array.

- **Example:**

```
arr = np.array([1, 2, 3, 4])
```

```
copy_arr = arr.copy()
```

```
view_arr = arr.view()
```

```
# Modify the original array
```

```
arr[0] = 100
```

```
print("Original Array:", arr) # [100, 2, 3, 4]
print("Copy Array:", copy_arr) # [1, 2, 3, 4]
print("View Array:", view_arr) # [100, 2, 3, 4]
```

### 4. Indexing & Slicing in 2D Arrays

#### a) Indexing in 2D Arrays

- You can access elements in a 2D array by specifying the **row and column** index.
- **Syntax:**

```
arr[row, column]
```

- **Example:**

```
arr = np.array([[1, 2], [3, 4], [5, 6]])
print(arr[1, 1]) # Access element at 2nd row and 2nd column
```

- **Output:**

4

## b) Slicing in 2D Arrays

- You can slice a **sub-array** from a 2D array by specifying the row and column ranges.

- **Syntax:**

```
arr[start_row:end_row, start_col:end_col]
```

- **Example:**

```
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(arr[:2, 1:]) # Get a sub-array from the first two rows and last two columns
```

- **Output:**

[[2 3]

[5 6]]

## Conclusion

We explored different methods of creating arrays in NumPy (`np.zeros()`, `np.ones()`, `np.full()`, etc.) and how to manipulate them using reshaping, copying, and slicing techniques. These operations allow for efficient data handling, which is crucial in data science tasks!

## Mini Project 1: Create a Data Table for a Sales Report

In this project, we will create a 2D NumPy array to represent a sales report and manipulate the data using different techniques.

### Step 1: Import NumPy

Start by importing the NumPy library.

```
import numpy as np
```

### Step 2: Create a 2D Array with Sales Data

Let's use `np.zeros()` to create an array representing sales for 3 products over 4 quarters. Initially, we'll set all values to 0.

```
sales_data = np.zeros((3, 4), dtype=int)
print(sales_data)
```

- **Output:** [[0 0 0 0]  
 [0 0 0 0]  
 [0 0 0 0]]

### Step 3: Fill the Sales Data with Values

Now, we can fill the array with sales data for each product across different quarters using `np.full()`.

```
sales_data = np.full((3, 4), [100, 150, 200, 250], dtype=int)
print(sales_data)
```

- **Output:** [[100 150 200 250]  
 [100 150 200 250]  
 [100 150 200 250]]

## Step 4: Reshape the Data Array

Let's reshape the array to simulate a different view of the data using .reshape().

```
reshaped_sales_data = sales_data.reshape(4, 3)
print(reshaped_sales_data)
```

- **Output:** [[100 150 200]
 [250 100 150]
 [200 250 100]
 [150 200 250]]

## Step 5: Generate Random Sales Data

We can use random data for sales in quarters using np.random.randint() to simulate a more realistic sales dataset.

```
random_sales_data = np.random.randint(50, 300, size=(3, 4))
print(random_sales_data)
```

- **Output (example):** [[101 275 96 235]
 [270 87 171 133]
 [ 68 255 156 86]]

## Step 6: Copying and Cloning Data

Use copy() to clone the array so that we can manipulate a copy without affecting the original.

```
sales_data_copy = random_sales_data.copy()
sales_data_copy[0, 0] = 999 # Modify the copy
print("Original Sales Data:\n", random_sales_data)
print("Modified Sales Data Copy:\n", sales_data_copy)
```

- **Output:** Original Sales Data:

[[101 275 96 235]

[270 87 171 133]

[ 68 255 156 86]]

Modified Sales Data Copy:

[[999 275 96 235]

[270 87 171 133]

[ 68 255 156 86]]

### **Step 7: Indexing and Slicing 2D Array**

Now, let's slice the 2D array to get data for specific products or quarters. For example, select sales data for the first two products across all quarters.

```
sliced_data = random_sales_data[:2, :]
print(sliced_data)
```

- **Output:** [[101 275 96 235]

[270 87 171 133]]

### **Mini Project 2: Simulation of Temperature Data for Multiple Cities**

In this project, we'll simulate temperature data for several cities and analyze it using NumPy.

#### **Step 1: Import NumPy**

```
import numpy as np
```

## Step 2: Create an Array of Temperatures Using np.ones()

We can create an initial temperature array with constant values (e.g., 25°C for each city during a month).

```
temperature_data = np.ones((5, 30), dtype=int) * 25
print(temperature_data)
```

- **Output:** [[25 25] [25 25] [25 25] [25 25] [25 25]]

## Step 3: Modify Data Using np.full()

Now let's assume that temperatures for different cities fluctuate, so we change the array to simulate real-world temperatures.

```
temperature_data = np.full((5, 30), 25)
temperature_data[0] = np.random.randint(20, 35, size=30) # City 1 temperatures
temperature_data[1] = np.random.randint(10, 20, size=30) # City 2 temperatures
print(temperature_data)
```

- **Output (random example):** [[32 28 30 21 33 27 24 22 30 31 34 21 27 32 26 21 23 34 29 23 22 30 33 24 29 28 31 24 33 32] [13 15 14 17 19 12 16 13 11 18 19 15 17 14 10 15 16 19 18 12 18 13 11 14 12 13 17 14 15 13]]

```
[25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25  
25 25 25 25 25]  
[25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25  
25 25 25 25 25]  
[25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25  
25 25 25 25 25]]
```

#### Step 4: Reshape the Data

```
reshape_temparature_data = temparature_data.reshape(30, 5)  
print(reshape_temparature_data)
```

#### Step 5: Indexing and Slicing

```
slicing = reshape_temparature_data[2:6 , :]  
print(slicing)
```

### Conclusion

In these two mini projects, we explored creating arrays, manipulating data, reshaping arrays, and indexing/slicing in NumPy. These skills are crucial for efficiently handling data, and they're widely used in real-world data analysis and machine learning tasks!

### Day 41 tasks

1. Create a 3x3 matrix filled with zeros using np.zeros() and display the matrix.
2. Create a 5x5 matrix filled with ones using np.ones() and set the data type to float.
3. Create a 4x4 matrix filled with the number 7 using np.full().

4. Generate a 4x4 identity matrix using `np.eye()`.
5. Create a 1D array of integers from 10 to 50 using `np.arange()` with a step of 5.
6. Create a 1D array of 100 equally spaced numbers between 0 and 1 using `np.linspace()`.
7. Generate a 3x3 matrix of random floating-point numbers between 0 and 1 using `np.random.rand()`.
8. Generate a 3x3 matrix of random numbers from a standard normal distribution using `np.random.randn()`.
9. Create a 3x3 matrix of random integers between 10 and 50 using `np.random.randint()`.
10. Reshape a 1D array of size 12 into a 3x4 matrix using `.reshape()` and display the reshaped matrix.
11. Flatten a 3x3 matrix into a 1D array using `.ravel()` and `.flatten()` and compare the results.
12. Clone an array using the `copy()` method and modify a value in the original array. Verify that the cloned array does not change.
13. Slice a 3x3 matrix to extract the second row and display it.

## Mini Project 1: Simulating Student Grades

### Objective:

Create and manipulate an array to simulate student grades across different subjects, and perform array manipulations to analyze and modify the data.

### Task:

1. Create a 5x4 array using `np.zeros()` to represent the grades of 5 students across 4 subjects. Initialize all grades to 0.

2. Fill the array with random grades between 50 and 100 using `np.random.randint()`.
3. Create a new array representing the maximum possible grades for each subject using `np.full()`.
4. Reshape the array of student grades to a 2D array representing grades for 2 semesters using `.reshape()`.
5. Clone the grades array and simulate an improvement in all student grades using the `copy()` method.
6. Extract and display the grades of a specific student (e.g., the second row) and a specific subject (e.g., the third column).

## Mini Project 2: Sales Data for Multiple Products

### Objective:

Simulate and manipulate an array representing sales data for multiple products over a week, and perform some operations to analyze the data.

### Task:

1. Create a  $3 \times 7$  array using `np.ones()` to represent sales data for 3 products across 7 days. Initialize all sales to 1 unit.
2. Modify the array to reflect more realistic sales data using `np.random.randint()`, with random sales numbers between 10 and 50.
3. Use `np.arange()` to generate an array of days of the week (1 to 7) and reshape it into a  $3 \times 7$  matrix to represent sales for 3 products.
4. Use `.ravel()` and `.flatten()` to compare how the array of sales data looks when flattened into 1D.
5. Clone the sales data array using the `copy()` method and modify the clone to simulate a discount and a boost in sales for all products.
6. Slice the array to extract and display the sales data for the first 3 days of the week for all products (first three columns).

# Day 42

## Array Operations & Mathematical Functions in NumPy

NumPy is highly optimized for performing numerical operations on large datasets. These operations include element-wise arithmetic operations, advanced mathematical functions, and aggregation functions that allow us to analyze data efficiently.

### 1. Element-wise Operations

#### Definition:

Element-wise operations are operations that are performed individually on each element of a NumPy array.

#### Syntax:

These operations use standard arithmetic operators like +, -, \*, /, etc.

#### Example:

```
import numpy as np
# Creating two arrays
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
# Element-wise addition
result = a + b
print(result) # Output: [5 7 9]
# Element-wise multiplication
result = a * b
print(result) # Output: [ 4 10 18]
```

In the example above, the addition and multiplication are applied element by element on the arrays a and b.

## 2. Broadcasting in NumPy

### Definition:

Broadcasting refers to how NumPy handles element-wise binary operations (such as addition, multiplication) on arrays of different shapes. When performing operations on arrays with different shapes, NumPy will automatically "broadcast" the smaller array to match the shape of the larger array.

### Syntax:

NumPy uses broadcasting rules internally when performing operations between arrays of different sizes or dimensions.

### Example:

```
import numpy as np

# A 3x1 array
a = np.array([[1], [2], [3]])

# A 1D array
b = np.array([10, 20, 30])

# Broadcasting happens here
result = a + b
print(result)
# Output:
# [[11 21 31]
#  [12 22 32]
#  [13 23 33]]
```

Here, the array a is broadcasted across the array b, and the operation is performed element-wise.

### 3. Universal Functions (ufunc) in NumPy

Universal functions, or ufuncs, are functions that operate element-wise on an array. They are highly optimized for performance.

#### Arithmetic Operations:

##### 1. np.add()

```
a = np.array([1, 2, 3])
```

```
b = np.array([4, 5, 6])
```

```
result = np.add(a, b)
```

```
print(result) # Output: [5 7 9]
```

##### 2. np.subtract()

```
result = np.subtract(a, b)
```

```
print(result) # Output: [-3 -3 -3]
```

##### 3. np.multiply()

```
result = np.multiply(a, b)
```

```
print(result) # Output: [ 4 10 18]
```

##### 4. np.divide()

```
result = np.divide(a, b)
```

```
print(result) # Output: [0.25 0.4 0.5]
```

##### 5. np.mod()

```
result = np.mod(a, b)
```

```
print(result) # Output: [1 2 3]
```

**Mathematical Functions:****1. np.exp()**

Computes the exponential of all elements in the input array.

```
a = np.array([1, 2, 3])
result = np.exp(a)
print(result) # Output: [ 2.71828183  7.3890561  20.08553692]
```

**2. np.sqrt()**

Computes the square root of each element in the array.

```
result = np.sqrt(a)
print(result) # Output: [1.        1.41421356 1.73205081]
```

**3. np.log()**

Computes the natural logarithm (base e) of each element.

```
result = np.log(a)
print(result) # Output: [0.        0.69314718 1.09861229]
```

**4. np.log10()**

Computes the base-10 logarithm of each element.

```
result = np.log10(a)
print(result) # Output: [0.        0.30103   0.47712125]
```

## 5. np.power()

Computes x raised to the power of y element-wise.

```
result = np.power(a, 2)
print(result) # Output: [1 4 9]
```

Trigonometric Functions:

### 1. np.sin()

```
a = np.array([0, np.pi/2, np.pi])
result = np.sin(a)
print(result) # Output: [0. 1. 0.]
```

### 2. np.cos()

```
result = np.cos(a)
print(result) # Output: [ 1.  0. -1.]
```

### 3. np.tan()

```
result = np.tan(a)
print(result) # Output: [ 0. 1.63312394e+16 -1.22464680e-16]
```

## 4. Aggregation Functions in NumPy

Aggregation functions help you compute summary statistics such as the sum, mean, variance, and more for NumPy arrays.

### **1. np.sum()**

Computes the sum of the elements.

```
a = np.array([1, 2, 3, 4, 5])
result = np.sum(a)
print(result) # Output: 15
```

### **2. np.mean()**

Computes the mean (average) of the elements.

```
result = np.mean(a)
print(result) # Output: 3.0
```

### **3. np.median()**

Computes the median of the elements.

```
result = np.median(a)
print(result) # Output: 3.0
```

### **4. np.var()**

Computes the variance of the elements.

```
result = np.var(a)
print(result) # Output: 2.0
```

### **5. np.std()**

Computes the standard deviation of the elements.

```
result = np.std(a)
print(result) # Output: 1.4142135623730951
```

## 6. np.min()

Computes the minimum value of the array.

```
result = np.min(a)
print(result) # Output: 1
```

## 7. np.max()

Computes the maximum value of the array.

```
result = np.max(a)
print(result) # Output: 5
```

## 8. np.argmin()

Finds the index of the minimum value.

```
result = np.argmin(a)
print(result) # Output: 0
```

## 9. np.argmax()

Finds the index of the maximum value.

```
result = np.argmax(a)
print(result) # Output: 4
```

## Summary:

- **Element-wise Operations:** Arithmetic operations performed element by element on arrays.
- **Broadcasting:** Automatically adjusts smaller arrays to match the size of larger arrays in arithmetic operations.
- **Universal Functions (ufuncs):** Functions like `np.add()`, `np.multiply()`, etc., that operate element-wise on arrays.
- **Mathematical Functions:** Functions like `np.exp()`, `np.sqrt()`, etc., that perform mathematical computations on arrays.
- **Aggregation Functions:** Functions like `np.sum()`, `np.mean()`, etc., that summarize the data in an array (e.g., sum, mean, min, max).

These features make NumPy a powerful tool for performing numerical computations efficiently on large datasets.

## Mini Project 1: Weather Data Analysis

### Objective:

Perform mathematical operations on weather data arrays representing temperature, humidity, and wind speed for different cities and days.

### Steps:

#### 1. Create the Arrays:

- a. Create a 3x7 array (3 cities, 7 days) representing the temperature ( $^{\circ}\text{C}$ ) for 3 cities over 7 days.
- b. Create a 3x7 array representing humidity (%).
- c. Create a 3x7 array representing wind speed (m/s).

#### 2. Element-wise Operations:

- a. Perform element-wise addition of the temperature and wind speed arrays to simulate the perceived temperature.
- b. Subtract the humidity from the temperature to see how humidity impacts perceived heat.

**3. Broadcasting:**

- a. Use broadcasting to subtract a global threshold temperature of 30°C from all the values in the temperature array to see which cities experienced a temperature above this threshold.

**4. Universal Functions (ufunc):**

- a. Apply np.sqrt() to the wind speed array to simulate the reduction of wind speed.
- b. Apply np.log10() to the humidity array to analyze the logarithmic trend of humidity over time.
- c. Use np.sin() to simulate the seasonal variations in temperature for each city.

**5. Aggregation Functions:**

- a. Calculate the mean and median temperatures across all cities for each day using np.mean() and np.median().
- b. Find the maximum wind speed across all cities using np.max() and the index of the highest wind speed using np.argmax().
- c. Calculate the variance and standard deviation of humidity for each city to understand how much it fluctuates across the week.

### **Mini Project: Weather Data Analysis**

**Objective:** Perform mathematical operations on weather data arrays (temperature, humidity, and wind speed) for different cities over a week.

## Step 1: Create the Arrays

We will create three 3x7 NumPy arrays representing:

- Temperature (°C)
- Humidity (%)
- Wind Speed (m/s)

### Code

```
import numpy as np
```

```
# Create random temperature data for 3 cities over 7 days (values between 20°C - 40°C)
```

```
temperature = np.random.randint(20, 41, size=(3, 7))
```

```
# Create random humidity data for 3 cities over 7 days (values between 50% - 100%)
```

```
humidity = np.random.randint(50, 101, size=(3, 7))
```

```
# Create random wind speed data for 3 cities over 7 days (values between 1 - 15 m/s)
```

```
wind_speed = np.random.randint(1, 16, size=(3, 7))
```

```
# Print the data
```

```
print("Temperature Data (°C):\n", temperature)
```

```
print("\nHumidity Data (%):\n", humidity)
```

```
print("\nWind Speed Data (m/s):\n", wind_speed)
```

## Step 2: Element-wise Operations

We will perform mathematical operations between arrays.

## 1. Add temperature and wind speed

This simulates perceived temperature (higher wind speed can increase perceived temperature).

```
perceived_temp = temperature + wind_speed
print("\nPerceived Temperature (°C):\n", perceived_temp)
```

## 2. Subtract humidity from temperature

This helps analyze the impact of humidity on perceived heat.

```
humidity_impact = temperature - humidity
print("\nTemperature After Humidity Effect:\n", humidity_impact)
```

## Step 3: Broadcasting

Broadcasting allows operations between arrays of different shapes.

### Subtract a global threshold (30°C) from the temperature array

This helps check which cities experienced a temperature above 30°C.

```
threshold = 30
above_threshold = temperature - threshold
print("\nTemperature Above 30°C:\n", above_threshold)
```

- Positive values → Above 30°C
- Zero or negative values → 30°C or below

## Step 4: Universal Functions (ufuncs)

NumPy ufuncs (universal functions) apply element-wise operations efficiently.

**1. Apply np.sqrt() on wind speed**

Simulates the reduction of wind speed effect.

```
wind_speed_reduction = np.sqrt(wind_speed)
print("\nReduced Wind Speed (sqrt applied):\n", wind_speed_reduction)
```

**2. Apply np.log10() on humidity**

Analyzes the logarithmic trend of humidity.

```
humidity_log = np.log10(humidity)
print("\nLogarithmic Trend of Humidity:\n", humidity_log)
```

**3. Apply np.sin() to simulate seasonal variations in temperature**

```
seasonal_temp_variation = np.sin(temperature)
print("\nSeasonal Temperature Variations:\n", seasonal_temp_variation)
```

**Step 5: Aggregation Functions**

Aggregation functions help analyze the overall weather trends.

**Calculate Mean & Median Temperature Across Cities (for each day)**

```
mean_temp = np.mean(temperature, axis=0)
median_temp = np.median(temperature, axis=0)
```

```
print("\nMean Temperature Across Cities (°C):\n", mean_temp)
print("\nMedian Temperature Across Cities (°C):\n", median_temp)
```

**Mean** → Average temperature.

**Median** → Middle temperature value (less affected by outliers).

**Find Maximum Wind Speed & Index of Highest Wind Speed**

```
max_wind_speed = np.max(wind_speed)
max_wind_speed_index = np.argmax(wind_speed)

print("\nMaximum Wind Speed (m/s):", max_wind_speed)
print("\nIndex of Highest Wind Speed:", max_wind_speed_index)
```

`np.argmax()` gives the index where the wind speed is highest.

**Calculate Variance & Standard Deviation of Humidity (for each city)**

```
humidity_variance = np.var(humidity, axis=1)
humidity_std_dev = np.std(humidity, axis=1)

print("\nHumidity Variance for Each City:\n", humidity_variance)
print("\nHumidity Standard Deviation for Each City:\n", humidity_std_dev)
```

**Variance** → How much humidity fluctuates.

**Standard Deviation** → Spread of humidity values.

**Summary of Analysis**

Operation	Function	Purpose
Add Wind Speed to Temperature	temperature + wind_speed	Simulate perceived temperature
Subtract Humidity from Temperature	temperature - humidity	Analyze humidity effect
Threshold Comparison	temperature - 30	Find days above 30°C

Square Root	<code>np.sqrt(wind_speed)</code>	Simulate wind speed reduction
Logarithm	<code>np.log10(humidity)</code>	Analyze humidity trend
Sin Function	<code>np.sin(temperature)</code>	Simulate seasonal temperature variation
Mean Temperature	<code>np.mean(temperature, axis=0)</code>	Find average temperature
Median Temperature	<code>np.median(temperature, axis=0)</code>	Find middle temperature value
Maximum Wind Speed	<code>np.max(wind_speed)</code>	Find highest wind speed
Index of Max Wind Speed	<code>np.argmax(wind_speed)</code>	Find where wind speed is highest
Humidity Variance	<code>np.var(humidity, axis=1)</code>	Check humidity fluctuation
Humidity Standard Deviation	<code>np.std(humidity, axis=1)</code>	Measure humidity spread

## Conclusion

- This project analyzes weather data using NumPy operations.
- It demonstrates element-wise operations, broadcasting, ufuncs, and aggregation.
- The approach is useful in climate analysis, weather prediction, and data science applications.

## Mini Project 2: Sales Data Analysis for a Retail Store

### Objective:

Analyze sales data for different products over several months and compute key statistics and operations like growth, average sales, and fluctuations.

**Steps:****1. Create the Arrays:**

- Create a 5x12 array representing sales data (in units) for 5 products over 12 months (columns represent months, rows represent products).
- Initialize the sales data with some random values using `np.random.randint()` to simulate sales in the range of 100-500 units.

**2. Element-wise Operations:**

- Calculate the yearly sales for each product by summing the data across each row using `np.sum()`.
- Calculate the monthly growth for each product by subtracting the sales from one month to the next using `np.diff()`.

**3. Broadcasting:**

- Apply a flat percentage discount (e.g., 10%) to all products by broadcasting a scalar value to the sales data array and multiplying.

**4. Universal Functions (ufunc):**

- Apply `np.exp()` to simulate exponential growth in sales for each product over time.
- Apply `np.log()` to the sales data to analyze sales on a logarithmic scale, giving insights into sales trends over the months.
- Apply `np.power()` to simulate sales projections based on previous sales for each product (e.g., projecting a 1.1x growth rate each month).

**5. Aggregation Functions:**

- Calculate the mean sales for all products over the 12 months using `np.mean()`.
- Calculate the variance of sales for each product using `np.var()` to understand fluctuations in monthly sales.
- Use `np.argmin()` and `np.argmax()` to find the months with the lowest and highest sales for each product.

- d. Calculate the standard deviation of sales across the months for each product to analyze the variability.

## **Summary:**

These mini projects will help you understand how to use element-wise operations, broadcasting, universal functions, and aggregation functions in NumPy. By simulating real-world scenarios such as weather and sales data analysis, you will gain hands-on experience in performing complex numerical operations on arrays.

## **Day 42 tasks**

### **1. Element-wise Operations**

- Create two 1D arrays of the same length and perform the following element-wise operations:
  - Addition
  - Subtraction
  - Multiplication
  - Division

### **2. Broadcasting in NumPy**

- Create a 2D array of shape (3, 3) and a 1D array of shape (3,). Use broadcasting to add the 1D array to each row of the 2D array.

### **3. Universal Function (ufunc): np.add()**

- Create two 1D arrays and use np.add() to perform element-wise addition. Verify that the result matches the addition operator (+).

**4. Universal Function (ufunc): np.subtract()**

- Create two 1D arrays and use np.subtract() to perform element-wise subtraction. Compare the result with the subtraction operator (-).

**5. Universal Function (ufunc): np.multiply()**

- Create two 1D arrays and use np.multiply() to perform element-wise multiplication. Compare the result with the multiplication operator (\*).

**6. Universal Function (ufunc): np.divide()**

- Create two 1D arrays and use np.divide() to perform element-wise division. Handle any potential division by zero cases by adding a small constant to the divisor.

**7. Universal Function (ufunc): np.mod()**

- Create two 1D arrays and use np.mod() to compute the element-wise modulus (remainder). Verify that the result matches the modulus operator (%).

**8. Mathematical Function: np.exp()**

- Create a 1D array and use np.exp() to compute the exponential ( $e^x$ ) of each element. Print the result.

**9. Mathematical Function: np.sqrt()**

- Create a 1D array and use np.sqrt() to compute the square root of each element. Handle any potential negative values by ensuring the array has non-negative values.

**10. Mathematical Function: np.log()**

- Create a 1D array with positive values and apply np.log() to compute the natural logarithm of each element.

**11. Mathematical Function: np.log10()**

- Create a 1D array and apply np.log10() to compute the base-10 logarithm of each element.

**12. Trigonometric Function: np.sin(), np.cos(), np.tan()**

- Create a 1D array representing angles in radians and apply np.sin(), np.cos(), and np.tan() to compute the trigonometric values for each angle.

**13. Aggregation Functions: np.sum(), np.mean(), np.min(), np.max()**

- Create a 2D array of random integers and compute:
  - The sum of all elements in the array.
  - The mean value of the array.
  - The minimum value in each column.
  - The maximum value in each row.

These tasks will help you practice a wide range of element-wise operations, broadcasting, universal functions, and aggregation functions in NumPy.

**Mini Project 1: Financial Data Analysis****Objective:**

Perform mathematical operations on financial data for multiple stocks and compute key statistics like average price, percentage change, and volatility.

**Steps:****1. Data Representation:**

- a. Create a 5x12 array representing the stock prices (in USD) for 5 stocks over 12 months (columns represent months, rows represent stocks).
- b. You can randomly generate the data using `np.random.randint()` to simulate stock prices between 100 and 500 USD.

**2. Element-wise Operations:**

- a. Calculate the percentage change in stock prices from one month to the next for each stock using element-wise subtraction and division.

**3. Broadcasting:**

- a. Subtract the **average stock price** across all months (row-wise) from each stock's price (broadcast the mean to all elements).

**4. Universal Functions (ufunc):**

- a. Use `np.sqrt()` to calculate the volatility (standard deviation) of stock prices for each stock (apply to each row).
- b. Use `np.exp()` to simulate price growth over time for each stock.
- c. Apply `np.log10()` to determine the logarithmic scale of stock prices over time.

**5. Aggregation Functions:**

- a. Compute the mean and median stock price across all months for each stock using `np.mean()` and `np.median()`.
- b. Find the maximum and minimum stock price across all months for each stock using `np.max()` and `np.min()`.
- c. Calculate the variance and standard deviation to analyze price fluctuations across the months for each stock.

## Mini Project 2: Exam Score Analysis

### Objective:

Analyze exam scores of students across multiple subjects and compute statistics like average score, best/worst scores, and overall performance.

### Steps:

#### 1. Data Representation:

- a. Create a 10x5 array representing the exam scores (out of 100) for 10 students in 5 subjects (columns represent subjects, rows represent students).
- b. Use `np.random.randint()` to generate random scores between 40 and 100 for each student and subject.

#### 2. Element-wise Operations:

- a. Calculate the total score for each student by summing their scores across all subjects using `np.sum()`.
- b. Calculate the percentage for each student by dividing their total score by the maximum possible score (e.g., 500 for 5 subjects).

#### 3. Broadcasting:

- a. Add a bonus score (e.g., 5 points) to each student's score across all subjects using broadcasting.

#### 4. Universal Functions (ufunc):

- a. Use `np.power()` to simulate the increase in performance by raising the percentage of each student to a power (e.g., a factor of 1.1).
- b. Use `np.log()` to determine the logarithmic growth of the average score across subjects for each student.

#### 5. Aggregation Functions:

- a. Compute the mean and median of the scores for each subject (column-wise).
- b. Find the best (maximum) and worst (minimum) performing student across all subjects using `np.max()` and `np.argmin()`.

- c. Calculate the variance and standard deviation of the scores across all subjects for each student to evaluate overall consistency.

## Day 43

### Advanced Indexing & Boolean Masking in NumPy

NumPy provides powerful indexing techniques that allow you to select, filter, and manipulate array elements based on conditions, rather than relying only on simple slicing. These techniques include fancy indexing, boolean masking, and various functions that allow for conditional selection and sorting. Below is a detailed explanation of these concepts with examples.

#### 1. Fancy Indexing

Fancy indexing allows you to index arrays using other arrays or lists, instead of simple integer indexing.

##### Definition:

- Fancy indexing is when we index an array using a list or another array of indices.
- This allows you to select elements in non-contiguous blocks and reorder the array.

##### Syntax:

array[indices]

##### Example:

```
import numpy as np
```

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])  
indices = [0, 2, 4, 6]
```

```
selected_elements = arr[indices]
print(selected_elements)
```

**Output:**

```
[1 3 5 7]
```

In this example, we select elements from indices 0, 2, 4, and 6 from the original array.

## 2. Boolean Masking

Boolean masking allows you to select elements of an array based on a condition.

**Definition:**

- A Boolean mask is an array of boolean values (True or False) that can be used to filter elements in a NumPy array.
- The condition is evaluated element-wise, and only the elements where the condition is True are returned.

**Syntax:**

```
array[condition]
```

**Example:**

```
import numpy as np
```

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
mask = arr > 5 # Boolean mask where elements are greater than 5
selected_elements = arr[mask]
print(selected_elements)
```

**Output:**

```
[6 7 8 9]
```

In this example, the Boolean mask filters all elements greater than 5.

### 3. Filtering Elements

Filtering elements is similar to Boolean masking, but it's done through more complex conditions.

**Definition:**

- You can apply filters (conditions) on an array and retrieve only the elements that satisfy those conditions.

**Syntax:**

```
array[condition]
```

**Example:**

```
import numpy as np

arr = np.array([10, 20, 30, 40, 50])
filtered = arr[arr > 30] # Filter elements greater than 30
print(filtered)
```

**Output:**

```
[40 50]
```

In this example, only elements greater than 30 are selected from the array.

## 4. Conditional Selection (`np.where()`, `np.select()`)

Conditional selection allows you to choose elements in an array based on a condition or multiple conditions.

### `np.where()`:

- `np.where()` returns the indices of the elements that satisfy a condition.

### Syntax:

`np.where(condition, x, y)`

- condition: The condition to test.
- x: Value to use when the condition is True.
- y: Value to use when the condition is False.

### Example:

```
import numpy as np
```

```
arr = np.array([1, 2, 3, 4, 5])
result = np.where(arr % 2 == 0, "even", "odd")
print(result)
```

### Output:

```
['odd' 'even' 'odd' 'even' 'odd']
```

In this example, `np.where()` checks if the element is even or odd and assigns labels accordingly.

### `np.select()`:

- `np.select()` allows you to specify multiple conditions and corresponding outcomes.

**Syntax:**

```
np.select(conditions, choices, default)
```

- conditions: List of conditions.
- choices: List of corresponding values for each condition.
- default: Value to return if no conditions are met.

**Example:**

```
import numpy as np
```

```
arr = np.array([10, 20, 30, 40, 50])
conditions = [arr < 20, (arr >= 20) & (arr <= 40), arr > 40]
choices = ['Low', 'Medium', 'High']
result = np.select(conditions, choices, default="Unknown")
print(result)
```

**Output:**

```
['Low' 'Medium' 'Medium' 'Medium' 'High']
```

In this example, np.select() categorizes elements as "Low", "Medium", or "High" based on their value.

## 5. Using np.any() and np.all()

- **np.any():** Returns True if at least one element in the array is True.
- **np.all():** Returns True if all elements in the array are True.

**Syntax:**

```
np.any(array)  
np.all(array)
```

**Example:**

```
import numpy as np  
  
arr = np.array([1, 0, 3, 4])  
print(np.any(arr > 0)) # True, because at least one element is greater than 0  
print(np.all(arr > 0)) # False, because not all elements are greater than 0
```

**Output:**

```
True  
False
```

## 6. Sorting and Searching in Arrays

Sorting and searching functions help you arrange and find specific elements in an array.

**np.sort():**

- Sorts the elements of an array.

**Syntax:**

```
np.sort(array)
```

**Example:**

```
import numpy as np  
  
arr = np.array([5, 3, 8, 1, 4])
```

```
sorted_arr = np.sort(arr)
print(sorted_arr)
```

**Output:**

[1 3 4 5 8]

**np.argsort():**

- Returns the indices that would sort an array.

**Syntax:**

```
np.argsort(array)
```

**Example:**

```
arr = np.array([5, 3, 8, 1, 4])
sorted_indices = np.argsort(arr)
print(sorted_indices)
```

**Output:**

[3 1 4 0 2]

**np.unique():**

- Returns the unique elements of an array.

**Syntax:**

```
np.unique(array)
```

**Example:**

```
arr = np.array([1, 2, 2, 3, 4, 4, 5])
unique_elements = np.unique(arr)
print(unique_elements)
```

**Output:**

```
[1 2 3 4 5]
```

**np.searchsorted():**

- Finds the indices where elements should be inserted to maintain the order.

**Syntax:**

```
np.searchsorted(sorted_array, values)
```

**Example:**

```
arr = np.array([1, 2, 3, 4, 5])
indices = np.searchsorted(arr, [2, 4])
print(indices)
```

**Output:**

```
[1 3]
```

**Summary:**

- Fancy Indexing allows you to index arrays using arrays or lists of indices.
- Boolean Masking lets you filter array elements based on conditions.
- Conditional Selection (using `np.where()` and `np.select()`) lets you apply conditions and choose values accordingly.
- `np.any()` and `np.all()` help check whether conditions are met for any or all elements of an array.
- Sorting functions (`np.sort()`, `np.argsort()`, `np.unique()`, `np.searchsorted()`) help organize and search through arrays efficiently.

These operations make NumPy powerful for data manipulation, especially when dealing with large datasets where conditional filtering and sorting are required.

## Mini Project 1: Student Grades Analysis

Goal: You are tasked with analyzing student grades for a class, identifying students who passed or failed, and categorizing their performance based on different conditions.

### Step-by-step Implementation:

1. **Create a NumPy Array for Student Grades:** Start by creating a NumPy array with student grades.

```
import numpy as np
```

```
# List of grades for 10 students
grades = np.array([85, 72, 90, 58, 67, 88, 95, 62, 73, 50])
```

2. **Use Boolean Masking to Identify Passing Students:** We will consider a grade of 60 or above as passing.

```
pass_mask = grades >= 60
passing_grades = grades[pass_mask]
print("Passing Grades:", passing_grades)
```

### Output:

```
Passing Grades: [85 72 90 67 88 95 62 73]
```

3. **Fancy Indexing to Get Specific Student Grades:** Select grades of specific students (e.g., students at positions 2, 5, and 8).

```
student_indices = [2, 5, 8]
selected_grades = grades[student_indices]
print("Selected Grades:", selected_grades)
```

**Output:**

Selected Grades: [90 88 73]

4. **Conditional Selection with np.where():** Create a list where "Pass" is assigned for grades above 60 and "Fail" otherwise.

```
result = np.where(grades >= 60, "Pass", "Fail")
print("Student Results:", result)
```

**Output:**

Student Results: ['Pass' 'Pass' 'Pass' 'Fail' 'Fail' 'Pass' 'Pass' 'Fail' 'Pass' 'Fail']

5. **Using np.any() and np.all():** Check if any student failed, and if all students passed.

```
any_failed = np.any(grades < 60)
all_passed = np.all(grades >= 60)

print("Any student failed:", any_failed)
print("Did all students pass?", all_passed)
```

**Output:**

Any student failed: True  
Did all students pass? False

**6. Sorting the Grades:** Sort the grades in ascending order.

```
sorted_grades = np.sort(grades)
print("Sorted Grades:", sorted_grades)
```

**Output:**

Sorted Grades: [50 58 62 67 72 73 85 88 90 95]

**7. Use np.argsort() to Find Sorted Indices:** Retrieve the indices that would sort the grades array.

```
sorted_indices = np.argsort(grades)
print("Sorted Indices:", sorted_indices)
```

**Output:**

Sorted Indices: [9 3 7 4 1 8 0 5 2 6]

**8. Finding Unique Grades:** Get the unique grades.

```
unique_grades = np.unique(grades)
print("Unique Grades:", unique_grades)
```

**Output:**

Unique Grades: [50 58 62 67 72 73 85 88 90 95]

## **Mini Project 2: E-commerce Product Price Analysis**

Goal: You are tasked with analyzing product prices for an online store and performing filtering, sorting, and conditional operations to make data-driven decisions.

**Step-by-step Implementation:**

- Create an Array for Product Prices:** Create a NumPy array containing the prices of products in your e-commerce store.

```
import numpy as np
```

```
# Product prices in USD
```

```
prices = np.array([199, 299, 99, 150, 249, 379, 89, 199, 259, 129])
```

- Fancy Indexing to Get Prices of Specific Products:** Use fancy indexing to get the prices of products in positions 1, 3, and 7.

```
product_indices = [1, 3, 7]
selected_prices = prices[product_indices]
print("Selected Product Prices:", selected_prices)
```

**Output:**

Selected Product Prices: [299 150 199]

- Boolean Masking to Filter Products Below a Certain Price:** Find products that are priced below \$200.

```
cheap_mask = prices < 200
cheap_products = prices[cheap_mask]
print("Cheap Products (below $200):", cheap_products)
```

**Output:**

Cheap Products (below \$200): [99 150 89 129]

- 4. Conditional Selection with np.select():** Classify products into "Budget", "Mid-range", and "Premium" based on their prices.

```
conditions = [prices < 150, (prices >= 150) & (prices <= 250), prices > 250]
choices = ['Budget', 'Mid-range', 'Premium']
price_category = np.select(conditions, choices, default="Unknown")
print("Product Price Categories:", price_category)
```

**Output:**

Product Price Categories: ['Budget' 'Premium' 'Budget' 'Budget' 'Mid-range'  
 'Premium' 'Budget' 'Budget' 'Mid-range' 'Budget']

- 5. Using np.any() and np.all() to Check Product Price Conditions:** Check if any product is "Premium" and if all products are under \$500.

```
any_premium = np.any(prices > 250)
all_under_500 = np.all(prices < 500)
```

```
print("Any Premium Products:", any_premium)
print("All products under $500:", all_under_500)
```

**Output:**

Any Premium Products: True  
 All products under \$500: True

- 6. Sorting the Prices:** Sort the prices in ascending order to help identify the cheapest products.

```
sorted_prices = np.sort(prices)
print("Sorted Product Prices:", sorted_prices)
```

**Output:**

Sorted Product Prices: [ 89 99 129 150 199 199 249 259 299 379]

7. **Finding the Indices of Sorted Prices:** Get the indices that would sort the array of product prices.

```
sorted_indices = np.argsort(prices)  
print("Sorted Indices of Products:", sorted_indices)
```

**Output:**

Sorted Indices of Products: [6 2 9 3 0 7 4 8 1 5]

8. **Finding Unique Product Prices:** Get the unique prices in the list.

```
unique_prices = np.unique(prices)  
print("Unique Product Prices:", unique_prices)
```

**Output:**

Unique Product Prices: [ 89 99 129 150 199 249 259 299 379]

### **Summary of Techniques Used:**

1. Fancy Indexing: Used to select elements at specific indices.
2. Boolean Masking: Used to filter elements based on a condition.
3. Conditional Selection: Used to classify products or grades based on conditions with `np.where()` and `np.select()`.
4. Sorting: Sorted product prices using `np.sort()` and found the sorted indices with `np.argsort()`.
5. Finding Unique Elements: Used `np.unique()` to find unique prices in the dataset.

6. Using np.any() and np.all(): Checked if any product was premium and if all products were under \$500.

## Day 43 tasks

1. Create a NumPy array of 15 random integers between 1 and 100.
  - a. Apply Boolean masking to select all values that are greater than 50.
2. Create a 1D NumPy array with 20 even numbers.
  - a. Use fancy indexing to select every third element from the array.
3. Create a 2D NumPy array of shape (4, 4) with values ranging from 1 to 16.
  - a. Use Boolean masking to find all elements greater than 10 in the array.
4. Create a 1D NumPy array of 10 elements.
  - a. Replace all values less than 5 with 0 using Boolean masking.
5. Given a 2D NumPy array of shape (6, 6), use fancy indexing to select rows 1, 3, and 5.
6. Create a 1D NumPy array of 10 random integers.
  - a. Use np.where() to assign "High" to values greater than 50 and "Low" otherwise.
7. Create a 2D array of shape (3, 3) with random integers between 10 and 50.
  - a. Find the index of the maximum value using np.argmax().
8. Create a 1D array of 12 values.
  - a. Use np.select() to assign "Small" for values less than 10, "Medium" for values between 10 and 20, and "Large" for values above 20.
9. Create a 1D array of 15 random numbers.
  - a. Use np.any() to check if any element in the array is negative.
10. Create a 1D array of 10 values between 0 and 100.
  - a. Use np.sort() to sort the array in descending order.
11. Create a 2D array of shape (5, 5) filled with numbers from 1 to 25.
  - a. Use fancy indexing to select the first 3 rows and columns 2 to 4.
12. Create a 1D NumPy array with 10 random values between 0 and 1.

- a. Use `np.searchsorted()` to find the index where a specific value would fit in the sorted array.
13. Create a 1D NumPy array of 8 values.
- a. Find the unique values in the array using `np.unique()` and sort them.

## Mini Project 1: Temperature Data Analysis

### Objective:

You are given a dataset containing daily temperature readings over a month. Your task is to filter, sort, and categorize the data based on specific temperature ranges.

### Steps:

1. Create a 1D NumPy array of 30 random float values representing daily temperatures in Celsius, ranging between -5 and 40.
2. Use Boolean masking to filter out all temperatures that are below 0°C (frosty days).
3. Use `np.where()` to categorize the temperatures as "Cold" (below 10°C), "Moderate" (between 10°C and 25°C), and "Hot" (above 25°C).
4. Find the highest temperature recorded using `np.argmax()` and the lowest temperature using `np.argmin()`.
5. Sort the temperature array in ascending order using `np.sort()`.
6. Check if any day had a temperature above 35°C using `np.any()`.
7. Use fancy indexing to select the first 7 days of temperatures (week 1).
8. Find where a temperature of 15°C would fit in the sorted array using `np.searchsorted()`.
9. Calculate the unique temperature values using `np.unique()` and count how many distinct temperatures exist.

## Mini Project 2: Product Rating Analysis

### Objective:

You are given a dataset of ratings for products in an online store. You need to filter, categorize, and sort the ratings to identify products with the best and worst ratings.

### Steps:

1. Create a 1D NumPy array of 50 random ratings between 1 and 5 (representing customer ratings for different products).
2. Use Boolean masking to find all products that have ratings greater than 3 (high-rated products).
3. Use np.select() to categorize the products into "Low Rating" (1-2), "Medium Rating" (3), and "High Rating" (4-5).
4. Find the index of the highest-rated product using np.argmax().
5. Sort the ratings array in descending order using np.sort().
6. Use np.any() to check if there are any products with a rating of 1.
7. Use fancy indexing to select the last 10 ratings in the array (for the last 10 products).
8. Use np.searchsorted() to find where a rating of 3.5 would fit in the sorted array.
9. Find the unique ratings and count how many distinct rating values exist using np.unique().

## Day 44

### Working with Multi-Dimensional Arrays in NumPy

NumPy provides powerful tools to work with multi-dimensional arrays (arrays with more than one axis). Here, we'll explore how to stack and split arrays, transpose arrays, swap axes, and work with 3D arrays. Let's break down the concepts and syntax in detail, followed by examples.

#### 1. Stacking Arrays:

Stacking means combining multiple arrays along a new axis (dimension). You can stack arrays vertically (row-wise), horizontally (column-wise), or depth-wise (for 3D arrays).

#### Syntax for Stacking Arrays:

- `np.vstack()`: Stacks arrays vertically (along rows).
- `np.hstack()`: Stacks arrays horizontally (along columns).
- `np.vstack()`: Stacks arrays depth-wise (along the third axis).

#### Examples:

##### Example 1: Using `np.vstack()`

Stacking two 1D arrays vertically (stacking along rows).

```
import numpy as np

# Create two 1D arrays
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Stack vertically
result = np.vstack((arr1, arr2))
print(result)
```

**Output:**

```
[[1 2 3]
 [4 5 6]]
```

Here, arr1 and arr2 are stacked on top of each other (along the vertical axis).

**Example 2: Using np.hstack()**

Stacking two 1D arrays horizontally (stacking along columns).

```
# Stack horizontally
result = np.hstack((arr1, arr2))
print(result)
```

**Output:**

```
[1 2 3 4 5 6]
```

Here, arr1 and arr2 are stacked side by side (along the horizontal axis).

**Example 3: Using np.vstack()**

Stacking two 2D arrays depth-wise (along the third axis).

```
# Create two 2D arrays
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])
```

```
# Stack depth-wise
result = np.vstack((arr1, arr2))
print(result)
```

**Output:**

```
[[[1 5]
 [2 6]]
```

```
[[3 7]
 [4 8]]]
```

Here, arr1 and arr2 are stacked along a third axis (depth), creating a 3D array.

## 2. Splitting Arrays:

Splitting refers to dividing an array into multiple smaller arrays along a specified axis.

### Syntax for Splitting Arrays:

- np.split(): Splits an array into multiple sub-arrays.
- np.array\_split(): Similar to np.split(), but it can split arrays into unequal parts.

### Examples:

#### Example 1: Using np.split()

Splitting a 1D array into 3 equal parts.

```
# Create a 1D array
arr = np.array([1, 2, 3, 4, 5, 6])

# Split into 3 parts
result = np.split(arr, 3)
print(result)
```

### Output:

```
[array([1, 2]), array([3, 4]), array([5, 6])]
```

The array is split into 3 equal parts.

**Example 2: Using np.array\_split()**

Splitting a 1D array into unequal parts.

```
# Create a 1D array
arr = np.array([1, 2, 3, 4, 5, 6])
```

```
# Split into 4 parts (unequal)
result = np.array_split(arr, 4)
print(result)
```

**Output:**

```
[array([1, 2]), array([3, 4]), array([5]), array([6])]
```

Here, the array is split into unequal parts.

**3. Transposing and Swapping Axes:**

- Transposing (.T): Inverts the dimensions of a matrix (rows become columns and vice versa).
- np.transpose(): Similar to .T, but allows you to specify more complex permutations of axes.
- np.swapaxes(): Swaps two axes of an array.

**Examples:****Example 1: Using .T (Transpose)**

Transposing a 2D array.

```
# Create a 2D array
arr = np.array([[1, 2, 3], [4, 5, 6]])
```

```
# Transpose using .T
```

```
result = arr.T  
print(result)
```

**Output:**

```
[[1 4]  
 [2 5]  
 [3 6]]
```

Here, the rows become columns and vice versa.

**Example 2: Using np.transpose()**

Transposing a 2D array with specific axes.

```
# Transpose with specific axes  
result = np.transpose(arr, (1, 0))  
print(result)
```

**Output:**

```
[[1 4]  
 [2 5]  
 [3 6]]
```

This is the same as using .T, but np.transpose() allows for more complex axis manipulations.

**Example 3: Using np.swapaxes()**

Swapping axes of a 3D array.

```
# Create a 3D array  
arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
```

```
# Swap axes 0 and 1
result = np.swapaxes(arr, 0, 1)
print(result)
```

**Output:**

```
[[[1 2]
 [5 6]]]
```

```
[[3 4]
 [7 8]]]
```

Here, the 0th and 1st axes of the array are swapped.

**4. Working with 3D Arrays:**

A 3D array can be thought of as an array of 2D arrays (like a matrix of matrices). You can manipulate and index 3D arrays similarly to how you work with 2D arrays, but with an additional axis.

**Example:****Example 1: Creating and Accessing Elements in a 3D Array**

```
# Create a 3D array
arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
```

```
# Access element from first "matrix"
result = arr[0, 1, 0] # Accessing element at index (0, 1, 0)
print(result)
```

**Output:**

3

### Example 2: Slicing in a 3D Array

```
# Slice the 3D array  
result = arr[0, :, :]  
print(result)
```

#### Output:

```
[[1 2]  
 [3 4]]
```

Here, we select the first "matrix" (2D slice) of the 3D array.

#### Conclusion:

Working with multi-dimensional arrays in NumPy allows you to manipulate data in highly flexible ways. You can easily stack, split, transpose, swap axes, and access elements in 1D, 2D, and 3D arrays. Mastering these techniques will make it easier to handle complex datasets and perform advanced data operations.

### Mini Project 1: Image Data Preprocessing using Stacking & Splitting Arrays

In this project, we will use stacking and splitting techniques to manipulate image data. We will create two 2D arrays to represent grayscale images and perform vertical and horizontal stacking. Then, we will split the images into smaller parts to simulate image segmentation.

#### Steps:

1. Install NumPy (if not already installed):

```
pip install numpy
```

2. Create two 2D arrays representing grayscale images: Here, we will simulate images by creating two 2D arrays of random pixel values.
3. Perform Vertical Stacking using np.vstack(): Stack the images vertically to create a new larger image.
4. Perform Horizontal Stacking using np.hstack(): Stack the images horizontally to form a new image.
5. Split the stacked images using np.split(): Split the large images into smaller parts for processing or analysis.

**Implementation:**

```
import numpy as np

# Step 1: Create two 2D arrays (representing grayscale images)
image1 = np.random.randint(0, 256, (5, 5)) # Image 1: 5x5 pixels
image2 = np.random.randint(0, 256, (5, 5)) # Image 2: 5x5 pixels

print("Image 1 (5x5):\n", image1)
print("Image 2 (5x5):\n", image2)

# Step 2: Perform vertical stacking (vstack)
stacked_vertical = np.vstack((image1, image2))
print("\nStacked Image Vertically (10x5):\n", stacked_vertical)

# Step 3: Perform horizontal stacking (hstack)
stacked_horizontal = np.hstack((image1, image2))
print("\nStacked Image Horizontally (5x10):\n", stacked_horizontal)

# Step 4: Split the stacked image into smaller parts
split_vertical = np.split(stacked_vertical, 2)
print("\nSplit Vertical Stacked Image:\n", split_vertical)
```

```
split_horizontal = np.split(stacked_horizontal, 2, axis=1)
print("\nSplit Horizontal Stacked Image:\n", split_horizontal)
```

### Explanation:

- **Step 1:** Two 5x5 images are created using `np.random.randint()` with pixel values between 0 and 255.
- **Step 2:** The images are stacked vertically to form a 10x5 array using `np.vstack()`.
- **Step 3:** The images are stacked horizontally to form a 5x10 array using `np.hstack()`.
- **Step 4:** We use `np.split()` to divide the stacked images back into parts (two halves in each case).

### Expected Output:

Image 1 (5x5):

```
[[ 35 134 61 198 215]
 [ 74 39 101 67 104]
 [227 91 206 75 73]
 [ 73 158 11 183 11]
 [132 101 56 139 50]]
```

Image 2 (5x5):

```
[[ 45 60 72 211 143]
 [ 72 62 99 71 102]
 [139 96 48 69 147]
 [ 24 136 248 108 232]
 [ 81 228 55 62 189]]
```

Stacked Image Vertically (10x5):

```
[[ 35 134 61 198 215]
 [ 74 39 101 67 104]
```

```
[227 91 206 75 73]
[ 73 158 11 183 11]
[132 101 56 139 50]
[ 45 60 72 211 143]
[ 72 62 99 71 102]
[139 96 48 69 147]
[ 24 136 248 108 232]
[ 81 228 55 62 189]]
```

Stacked Image Horizontally (5x10):

```
[[ 35 134 61 198 215 45 60 72 211 143]
[ 74 39 101 67 104 72 62 99 71 102]
[227 91 206 75 73 139 96 48 69 147]
[ 73 158 11 183 11 24 136 248 108 232]
[132 101 56 139 50 81 228 55 62 189]]
```

Split Vertical Stacked Image:

```
[array([[ 35, 134, 61, 198, 215],
       [ 74, 39, 101, 67, 104],
       [227, 91, 206, 75, 73],
       [ 73, 158, 11, 183, 11],
       [132, 101, 56, 139, 50]]),
 array([[ 45, 60, 72, 211, 143],
       [ 72, 62, 99, 71, 102],
       [139, 96, 48, 69, 147],
       [ 24, 136, 248, 108, 232],
       [ 81, 228, 55, 62, 189]])]
```

Split Horizontal Stacked Image:

```
[array([[ 35, 134, 61, 198, 215],
       [ 74, 39, 101, 67, 104],
```

```
[227, 91, 206, 75, 73],  
[ 73, 158, 11, 183, 11],  
[132, 101, 56, 139, 50]]),  
array([[ 45, 60, 72, 211, 143],  
       [ 72, 62, 99, 71, 102],  
       [139, 96, 48, 69, 147],  
       [ 24, 136, 248, 108, 232],  
       [ 81, 228, 55, 62, 189]]])
```

## Mini Project 2: 3D Data Manipulation for Scientific Analysis

In this project, we will simulate scientific data stored in 3D arrays. We'll work with 3D arrays to represent data across time, space, and temperature. We will apply transposition, axis swapping, and slicing operations to manipulate and analyze the data.

### Steps:

1. Create a 3D array to represent scientific data (temperature over time and space).
2. Transpose the 3D array to switch axes.
3. Swap axes to change the orientation of the data.
4. Slice the 3D array to extract subsets of the data.

### Implementation:

```
import numpy as np
```

```
# Step 1: Create a 3D array representing scientific data (e.g., temperature)  
# Shape: (2, 3, 4) -> 2 time points, 3 regions, 4 temperature readings each  
data = np.random.randint(0, 100, (2, 3, 4))
```

```

print("Original 3D Array (2 time points, 3 regions, 4 temperature readings):\n",
data)

# Step 2: Transpose the 3D array (swap the axes)
transposed_data = np.transpose(data, (1, 0, 2))
print("\nTransposed 3D Array (3 regions, 2 time points, 4 temperature
readings):\n", transposed_data)

# Step 3: Swap axes (swap axis 0 and axis 1)
swapped_data = np.swapaxes(data, 0, 1)
print("\nSwapped Axes (3 regions, 2 time points, 4 temperature readings):\n",
swapped_data)

# Step 4: Slice the 3D array (extract data for the first time point)
first_time_point_data = data[0, :, :]
print("\nData for First Time Point (3 regions, 4 temperature readings):\n",
first_time_point_data)

```

### **Explanation:**

- **Step 1:** A 3D array of shape (2, 3, 4) is created to represent temperature data across two time points, three regions, and four readings per region.
- **Step 2:** The array is transposed using `np.transpose()` to swap the time and region axes.
- **Step 3:** The axes are swapped with `np.swapaxes()` to change the orientation of the data.
- **Step 4:** A slice is taken from the 3D array to extract data for the first time point.

### **Expected Output:**

Original 3D Array (2 time points, 3 regions, 4 temperature readings):  
[[[ 5 97 18 62]

[95 76 81 72]

[62 59 71 53]]

[[77 61 71 31]

[43 74 23 92]

[47 40 99 21]]]

Transposed 3D Array (3 regions, 2 time points, 4 temperature readings):

[[[ 5 97 18 62]

[77 61 71 31]]

[[95 76 81 72]

[43 74 23 92]]

[[62 59 71 53]

[47 40 99 21]]]

Swapped Axes (3 regions, 2 time points, 4 temperature readings):

[[[ 5 97 18 62]

[77 61 71 31]]

[[95 76 81 72]

[43 74 23 92]]

[[62 59 71 53]

[47 40 99 21]]]

Data for First Time Point (3 regions, 4 temperature readings):

[[ 5 97 18 62]

[95 76 81 72]

[62 59 71 53]]

## Conclusion:

In these two mini-projects, we worked with multi-dimensional arrays to manipulate and analyze data using stacking, splitting, transposing, swapping axes, and slicing operations. These techniques are commonly used in fields like image processing, scientific data analysis, and machine learning, where multi-dimensional arrays are often the primary data structure.

## Day 44 tasks

### 1. Create two 2D arrays and stack them vertically using np.vstack()

- Create two arrays of shape (3, 4), then stack them vertically to create a (6, 4) array. Print the resulting array.

### 2. Horizontally stack two 2D arrays using np.hstack()

- Create two arrays of shape (3, 4), then stack them horizontally to create a (3, 8) array. Print the resulting array.

### 3. Create a 3D array and stack two 2D arrays along the third axis using np.dstack()

- Create two arrays of shape (3, 4), and stack them along the third axis to form a 3D array of shape (3, 4, 2).

### 4. Split a 2D array into two equal parts using np.split()

- Create a 2D array of shape (6, 4) and split it into two smaller arrays, each of shape (3, 4). Print the resulting arrays.

**5. Use np.array\_split() to split a 2D array into three parts**

- Create a 2D array of shape (7, 4), then split it into three parts along the first axis. The resulting arrays may have different sizes.

**6. Create a 3D array and transpose it using .T**

- Create a 3D array of shape (2, 3, 4), then transpose it using .T and print the new shape.

**7. Use np.transpose() to transpose a 3D array**

- Create a 3D array of shape (2, 3, 4), then transpose the axes so that the new shape becomes (4, 3, 2). Print the resulting array.

**8. Swap the axes of a 3D array using np.swapaxes()**

- Create a 3D array of shape (2, 3, 4), then swap axes 0 and 2, resulting in a shape of (4, 3, 2). Print the new array.

**9. Slice a 3D array to extract a 2D array from a specific index**

- Create a 3D array of shape (5, 3, 4), then slice it to extract the 2D array at index 2 along the first axis (e.g., shape (3, 4)).

**10. Create a 2D array and perform horizontal stacking with another array of different size**

- Create a 2D array of shape (3, 4) and another array of shape (3, 2), then perform horizontal stacking. Handle any size mismatch with broadcasting.

### **11. Stack multiple 1D arrays vertically to form a 2D array using np.vstack()**

- Create three 1D arrays of shape (4,) and stack them vertically to create a 2D array of shape (3, 4).

### **12. Stack two 2D arrays horizontally to create a larger 2D array using np.hstack()**

- Create two 2D arrays of shape (3, 3) and stack them horizontally to form a new array of shape (3, 6).

### **13. Split a 1D array into multiple parts using np.split() and analyze the results**

- Create a 1D array of size 12, then split it into four equal parts. Print the resulting subarrays and explain the output.

## **Mini Project 1: Student Grades Analysis**

### **Objective:**

You are tasked with analyzing the grades of students across different subjects over multiple terms.

### **Instructions:**

1. Create a 3D NumPy array `grades` with shape (6, 4, 3) representing the grades of 6 students over 4 terms in 3 subjects (Math, Science, English).
  - a. The first axis represents students (6).
  - b. The second axis represents terms (4).
  - c. The third axis represents subjects (3).
2. Use vertical stacking (`np.vstack()`) to combine the grades of two students into a single array.
3. Transpose the grades array to swap the axes and view the result in such a way that terms are now the first axis and students come last.

4. Split the grades array using `np.split()` to examine the grades of each subject individually. Each split should return data for one subject across all students and terms.
5. Use `np.swapaxes()` to swap the terms and subjects axes, and print the reshaped array to analyze how this affects the structure of the data.
6. Extract and print the grades for a specific student (say, the 3rd student) in a specific subject (e.g., Science) for all terms using slicing.

## Mini Project 2: Weather Data Analysis

### Objective:

You have a weather dataset that tracks temperature, humidity, and rainfall for different cities over several weeks.

### Instructions:

1. Create a 3D NumPy array `weather_data` of shape (5, 7, 3) representing the weather data of 5 cities, each tracked for 7 days across 3 parameters (Temperature, Humidity, Rainfall).
  - a. The first axis represents cities (5).
  - b. The second axis represents days (7).
  - c. The third axis represents weather parameters (3).
2. Use horizontal stacking (`np.hstack()`) to combine the weather data of two cities, forming a larger array. The resulting shape should be (5, 7, 6) — the weather data of 5 cities stacked side by side.
3. Transpose the `weather_data` array to swap the cities and days axes, resulting in a shape of (7, 5, 3). Print the resulting array to observe how the data is transformed.
4. Split the array into smaller chunks by using `np.array_split()` along the first axis (cities) to get the weather data of individual cities.

5. Swap axes of the `weather_data` array using `np.swapaxes()`, changing the arrangement of cities and days to see how the analysis changes when the axes are interchanged.
6. Use slicing to extract the temperature data for a specific day (e.g., Day 4) across all cities and print the result.

## Day 45

### NumPy Linear Algebra & Statistics

NumPy provides a variety of powerful tools for performing linear algebra operations and statistical calculations on arrays. Let's break down the major functions and concepts used in these areas.

#### 1. Matrix Operations in NumPy:

##### Matrix Multiplication:

Matrix multiplication is a fundamental operation in linear algebra. In NumPy, you can perform matrix multiplication using:

- `np.dot()`: Calculates the dot product of two arrays.
- `np.matmul()`: Performs matrix multiplication. It is equivalent to the `@` operator, introduced in Python 3.5.

##### Syntax:

- `np.dot(a, b)`
  - Multiplies two arrays `a` and `b`.
- `np.matmul(a, b)`
  - Also performs matrix multiplication but allows for matrix-vector multiplication and broadcasting.
- `@`

- The @ operator is shorthand for np.matmul() in Python.

**Example:**

```
import numpy as np
```

```
# 2D Arrays (Matrices)
```

```
A = np.array([[1, 2], [3, 4]])
```

```
B = np.array([[5, 6], [7, 8]])
```

```
# Matrix multiplication using np.dot()
```

```
result_dot = np.dot(A, B)
```

```
print(result_dot)
```

```
# Matrix multiplication using np.matmul() or @ operator
```

```
result_matmul = np.matmul(A, B)
```

```
print(result_matmul)
```

```
# Using @ operator
```

```
result_at = A @ B
```

```
print(result_at)
```

**Inverse of a Matrix:**

- **np.linalg.inv():** Computes the inverse of a square matrix. The inverse of a matrix A is denoted as  $A^{-1}A^{-1}$ , and it satisfies  $A \times A^{-1} = I$  where  $I$  is the identity matrix.

**Syntax:**

- **np.linalg.inv(a)**
  - Returns the inverse of matrix a.

### Determinant of a Matrix:

- **np.linalg.det():** Computes the determinant of a square matrix. The determinant is a scalar value that indicates if the matrix is invertible.

### Syntax:

- np.linalg.det(a)
  - Returns the determinant of matrix a.

### Example:

```
import numpy as np

# Square Matrix
A = np.array([[4, 7], [2, 6]])
```

```
# Inverse of Matrix A
A_inv = np.linalg.inv(A)
print("Inverse of A:\n", A_inv)
```

```
# Determinant of Matrix A
A_det = np.linalg.det(A)
print("Determinant of A:", A_det)
```

## 2. Eigenvalues and Eigenvectors:

Eigenvalues and eigenvectors are fundamental concepts in linear algebra. They are used in various applications, such as principal component analysis (PCA), solving systems of differential equations, and more.

### Eigenvalues & Eigenvectors:

- **np.linalg.eig():** This function computes the eigenvalues and right eigenvectors of a square matrix.

**Syntax:**

- `np.linalg.eig(a)`
  - Computes the eigenvalues and eigenvectors of the matrix `a`.
  - Returns a tuple (`eigenvalues, eigenvectors`).

**Example:**

```
import numpy as np
```

```
# Square Matrix
```

```
A = np.array([[4, -2], [1, 1]])
```

```
# Compute eigenvalues and eigenvectors
```

```
eigenvalues, eigenvectors = np.linalg.eig(A)
```

```
print("Eigenvalues:", eigenvalues)
```

```
print("Eigenvectors:\n", eigenvectors)
```

**3. Solving Linear Equations:**

Linear algebra plays a crucial role in solving systems of linear equations. If you have a system of equations represented as  $Ax=b$ , where  $A$  is a square matrix and  $b$  is a vector, you can use NumPy to solve for  $x$ .

**Solving Linear Equations:**

- `np.linalg.solve()`: Solves the system of linear equations  $Ax=b$ , where  $A$  is a square matrix and  $b$  is a vector.

**Syntax:**

- `np.linalg.solve(A, b)`
  - Solves the equation  $Ax=b$  for  $x$ .

**Example:**

```

import numpy as np

# Coefficient matrix A
A = np.array([[3, 1], [1, 2]])

# Constant matrix b
b = np.array([9, 8])

# Solve the system of equations
x = np.linalg.solve(A, b)

print("Solution to the system of equations:", x)

```

**4. Statistics & Probability in NumPy:**

NumPy provides many statistical and random number generation functions, which are essential for data analysis, simulation, and probabilistic models.

**Percentiles:**

- **np.percentile():** This function computes the nth percentile of a given data array. The nth percentile is the value below which n percent of the data falls.

**Syntax:**

- np.percentile(a, q)
  - a: Input array.
  - q: The percentile to compute, which can be a scalar or an array.

**Example:**

```
import numpy as np
```

```
# Array of data
data = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# 50th percentile (median)
percentile_50 = np.percentile(data, 50)
print("50th Percentile:", percentile_50)
```

**Histograms:**

- **np.histogram()**: Computes the histogram of a dataset, which is a representation of the frequency distribution of the dataset.

**Syntax:**

- np.histogram(a, bins)
  - a: Input array.
  - bins: The number of bins or the edges of the bins for the histogram.

**Example:**

```
import numpy as np
```

```
# Array of data
data = np.random.randn(1000)

# Compute the histogram with 10 bins
hist, bin_edges = np.histogram(data, bins=10)

print("Histogram:", hist)
print("Bin Edges:", bin_edges)
```

**Random Choice:**

- **np.random.choice()**: This function generates a random sample from a given 1D array. It is useful for simulating random selections or sampling.

**Syntax:**

- `np.random.choice(a, size, replace)`
  - `a`: The input array or list.
  - `size`: The number of samples to generate.
  - `replace`: Whether to allow sampling the same element multiple times (default is True).

**Example:**

```
import numpy as np
```

```
# Array of possible outcomes
outcomes = np.array([1, 2, 3, 4, 5, 6])

# Randomly select 3 outcomes with replacement
random_samples = np.random.choice(outcomes, size=3)
print("Random Samples:", random_samples)
```

**Summary of Key Functions:**

- **Matrix Operations:**
  - `np.dot()`, `np.matmul()`, `@`: Matrix multiplication.
  - `np.linalg.inv()`: Inverse of a matrix.
  - `np.linalg.det()`: Determinant of a matrix.
  - `np.linalg.eig()`: Eigenvalues and eigenvectors.
  - `np.linalg.solve()`: Solving linear equations.
- **Statistics & Probability:**
  - `np.percentile()`: Percentiles of data.
  - `np.histogram()`: Histogram of data.
  - `np.random.choice()`: Random sampling.

These functions make NumPy a powerful library for solving problems in linear algebra and statistics.

## **Mini Project 1: Solving Linear Equations and Matrix Operations**

**Objective:** Solve a system of linear equations, perform matrix multiplication, calculate eigenvalues and eigenvectors, and work with determinants and matrix inversion.

### **Step 1: Setup and Import Libraries**

```
import numpy as np
```

### **Step 2: Define the System of Linear Equations**

We will solve the system of equations  $Ax=b$  where:

- $A=[3\ 1\ 2]$
- $b=[9\ 8]$

Define the coefficient matrix A and the constant matrix b:

```
# Coefficient Matrix (A)
A = np.array([[3, 1], [1, 2]])
```

```
# Constant Matrix (b)
b = np.array([9, 8])
```

### **Step 3: Solve Linear Equations**

Use `np.linalg.solve()` to solve the equation  $Ax=b$ :

```
# Solve the system of linear equations
x = np.linalg.solve(A, b)
print("Solution to the system of equations:", x)
```

#### **Step 4: Matrix Operations**

- **Matrix Multiplication:** We will multiply matrices A and B:

```
# Define another matrix B
```

```
B = np.array([[1, 2], [3, 4]])
```

```
# Matrix multiplication using np.dot() and np.matmul()
```

```
result_dot = np.dot(A, B)
```

```
print("Matrix Multiplication using np.dot():\n", result_dot)
```

```
result_matmul = np.matmul(A, B)
```

```
print("Matrix Multiplication using np.matmul():\n", result_matmul)
```

- **Inverse of Matrix A:**

```
# Inverse of matrix A
```

```
A_inv = np.linalg.inv(A)
```

```
print("Inverse of Matrix A:\n", A_inv)
```

- **Determinant of Matrix A:**

```
# Determinant of Matrix A
```

```
A_det = np.linalg.det(A)
```

```
print("Determinant of Matrix A:", A_det)
```

#### **Step 5: Eigenvalues and Eigenvectors**

Use np.linalg.eig() to calculate the eigenvalues and eigenvectors of matrix A:

```
# Eigenvalues and Eigenvectors of A
eigenvalues, eigenvectors = np.linalg.eig(A)
print("Eigenvalues:", eigenvalues)
print("Eigenvectors:\n", eigenvectors)
```

**Step 6: Summarize Output**

The output should include:

- The solution to the system of equations  $Ax=b$
- Results of matrix multiplication.
- The inverse and determinant of A.
- Eigenvalues and eigenvectors of A.

**Mini Project 2: Statistical Analysis and Probability Simulations**

**Objective:** Perform basic statistical analysis (mean, percentile, histogram), and simulate random events using probability functions.

**Step 1: Setup and Import Libraries**

```
import numpy as np
```

**Step 2: Generate Random Data**

Use `np.random.randn()` to generate random data:

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

**Step 3: Basic Statistical Analysis**

- **Mean, Median, and Standard Deviation:**

```
# Mean of the data  
mean = np.mean(data)  
print("Mean of data:", mean)  
  
# Median of the data  
median = np.median(data)  
print("Median of data:", median)  
  
# Standard deviation of the data  
std_dev = np.std(data)  
print("Standard Deviation of data:", std_dev)
```

- **Percentiles:**

```
# Compute 25th, 50th (median), and 75th percentiles  
percentile_25 = np.percentile(data, 25)  
percentile_50 = np.percentile(data, 50)  
percentile_75 = np.percentile(data, 75)  
  
print("25th Percentile:", percentile_25)  
print("50th Percentile (Median):", percentile_50)  
print("75th Percentile:", percentile_75)
```

#### **Step 4: Create and Visualize a Histogram**

Use `np.histogram()` to visualize the frequency distribution of the data:

```
# Compute the histogram of the data  
hist, bin_edges = np.histogram(data, bins=20)  
  
print("Histogram:\n", hist)
```

```
print("Bin Edges:\n", bin_edges)
```

## Step 5: Simulate Random Events

Use `np.random.choice()` to simulate a random event, like selecting a random number from a predefined set of outcomes:

```
# Define possible outcomes
outcomes = np.array([1, 2, 3, 4, 5, 6])

# Randomly select 10 outcomes
random_samples = np.random.choice(outcomes, size=10)
print("Random Samples:", random_samples)
```

## Step 6: Summarize Output

The output will include:

- Mean, median, and standard deviation of the generated data.
- Percentiles (25th, 50th, 75th).
- Histogram of the data showing frequency distribution.
- Randomly selected outcomes based on the specified set.

## Summary of Both Projects:

- **Mini Project 1** focuses on solving linear equations, performing matrix operations (like multiplication, inversion, determinant), and working with eigenvalues and eigenvectors.
- **Mini Project 2** focuses on statistical analysis of random data, including calculating mean, median, percentiles, standard deviation, and visualizing

the data through histograms. Additionally, it simulates random events using the `np.random.choice()` function.

These projects will help you practice a wide range of important linear algebra and statistical operations in NumPy.

## Day 45 tasks

### 1. Matrix Multiplication using `np.dot()`:

- Create two random 3x3 matrices and multiply them using `np.dot()`.  
Display the resulting matrix.

### 2. Matrix Multiplication using `np.matmul()`:

- Create two random 3x3 matrices and multiply them using `np.matmul()`. Compare the result with the one from `np.dot()`.

### 3. Matrix Multiplication using the @ Operator:

- Create two 2x2 matrices and multiply them using the @ operator.  
Verify the result matches that of `np.dot()`.

### 4. Calculate the Inverse of a Matrix:

- Create a random 2x2 matrix and calculate its inverse using `np.linalg.inv()`. Verify that multiplying the matrix by its inverse results in the identity matrix.

### 5. Calculate the Determinant of a Matrix:

- Create a 3x3 matrix and calculate its determinant using `np.linalg.det()`. Interpret the result.

### 6. Find Eigenvalues and Eigenvectors:

- Create a 3x3 matrix and use `np.linalg.eig()` to calculate the eigenvalues and eigenvectors. Print the eigenvalues and eigenvectors of the matrix.

### 7. Solve a System of Linear Equations:

- Create a coefficient matrix A and a constant vector b. Use `np.linalg.solve()` to solve for the vector x in the equation  $Ax=b$

### 8. Calculate the Percentile of an Array:

- a. Create an array of 1000 random numbers and compute the 25th, 50th, and 75th percentiles using np.percentile().

**9. Create and Plot a Histogram:**

- a. Generate a random dataset of 1000 points using np.random.randn() and plot the histogram of the data. Use np.histogram() to calculate the histogram values.

**10. Simulate a Random Sample from a Dataset:**

- Create an array with values from 1 to 10. Use np.random.choice() to randomly select 5 numbers from this array. Repeat the operation 3 times to show different samples.

**11. Generate a Normal Distribution of Random Data:**

- Use np.random.randn() to generate 1000 random numbers following a standard normal distribution. Calculate and print the mean and standard deviation.

**12. Create a Cumulative Distribution:**

- Using the random data generated in the previous task, create and plot a cumulative distribution using the np.cumsum() function.

**13. Simulate a Coin Toss (Bernoulli Trial):**

- Use np.random.choice() to simulate 100 coin tosses (Heads = 1, Tails = 0). Calculate the proportion of heads and tails.

## Mini Project 1: Linear Equation Solver

**Objective:** Solve a system of linear equations using matrix operations.

**Description:**

1. Create a coefficient matrix A (3x3) and a constant vector b (3x1). The matrix A will represent the coefficients of three linear equations, and vector b will represent the constants on the right-hand side of the equations.
2. Use np.linalg.solve() to solve for the unknown variables xx in the equation  $A \times x = b$ .

3. Verify the solution by multiplying the matrix A with the solution vector x to check if it equals the vector b.

**Steps:**

- Generate random values for the elements of the matrix A and vector b.
- Use np.linalg.solve() to find the solution.
- Verify the result by performing matrix multiplication using np.dot() or np.matmul().

**Mini Project 2: Statistical Analysis of Random Data**

**Objective:** Perform statistical analysis on a dataset and visualize the results.

**Description:**

1. Generate a random dataset of 1000 values using np.random.randn(). This will simulate a standard normal distribution.
2. Calculate the following statistical metrics:
  - a. Mean, standard deviation, and variance using np.mean(), np.std(), and np.var().
  - b. Percentiles (25th, 50th, and 75th) using np.percentile().
3. Plot a histogram of the data using np.histogram() and visualize it using matplotlib or any other plotting library.
4. Use np.random.choice() to randomly select 100 values from the dataset and calculate the mean and standard deviation of the selected values.

**Steps:**

- Generate the dataset using np.random.randn().
- Calculate and display the statistical metrics.
- Use np.histogram() to calculate the histogram of the dataset and plot it.
- Use np.random.choice() to simulate random selection and perform analysis on the subset.

These tasks will give you hands-on experience with linear algebra operations and basic statistical methods in NumPy. Try implementing them and analyze the results!

## Day 46

### NumPy with Real-World Data & Performance Optimization

In this section, we'll explore how to work with real-world data using NumPy, handle missing values, and optimize performance using different techniques.

#### 1. Loading & Saving Data

##### a. np.loadtxt()

**Definition:** np.loadtxt() is used to load data from a text file. It reads the file line by line and parses the contents into a NumPy array.

##### Syntax:

```
numpy.loadtxt(fname, dtype=<data-type>, delimiter=<delimiter>,
skiprows=<number-of-rows-to-skip>, usecols=<columns-to-read>)
```

- fname: File path.
- dtype: Data type of the resulting array. (default: float)
- delimiter: Character separating values in the file.
- skiprows: Number of lines to skip from the start of the file.
- usecols: Columns to read from the file.

##### Example:

```
import numpy as np
```

```
# Example file: data.txt contains:
```

```
# 1 2 3
```

```
# 4 5 6  
# 7 8 9  
  
data = np.loadtxt('data.txt')  
print(data)
```

### b. np.genfromtxt()

**Definition:** np.genfromtxt() is more flexible than np.loadtxt(). It can handle missing data and custom data types.

#### Syntax:

```
numpy.genfromtxt(fname, delimiter=<delimiter>, dtype=<data-type>,  
skip_header=<number-of-lines>, filling_values=<value-to-fill-missing-data>)
```

- `filling_values`: Value used to fill missing data.
- `skip_header`: Number of lines to skip from the start.

#### Example:

```
import numpy as np
```

```
# Example file: data_with_missing_values.txt contains:
```

```
# 1 2 3  
# 4 NaN 6  
# 7 8 9
```

```
data = np.genfromtxt('data_with_missing_values.txt', delimiter=' ', filling_values=-  
1)  
print(data)
```

### c. np.savetxt()

**Definition:** np.savetxt() is used to save data from a NumPy array to a text file.

#### Syntax:

```
numpy.savetxt(fname, array, delimiter=<delimiter>, fmt=<format-specifier>)
```

- fname: Name of the file.
- array: The NumPy array to save.
- fmt: Format specifier for numbers (e.g., '%f' for floating-point).

#### Example:

```
import numpy as np
```

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
np.savetxt('output.txt', arr, delimiter=' ', fmt='%d')
```

### d. np.save() & np.load()

**Definition:** np.save() is used to save a NumPy array to a binary .npy file, and np.load() is used to load that file back into a NumPy array.

#### Syntax:

```
numpy.save(fname, array)
array = numpy.load(fname)
```

#### Example:

```
import numpy as np
```

```
arr = np.array([1, 2, 3, 4, 5])
np.save('data.npy', arr) # Save the array to a binary file
```

```
loaded_arr = np.load('data.npy') # Load the array from the binary file
print(loaded_arr)
```

## 2. Handling Missing Values

Missing values can be common in real-world datasets. NumPy provides ways to handle and fill missing values.

### a. np.nan and np.isnan()

**Definition:** np.nan is a special value in NumPy that represents "Not a Number" and is used to indicate missing or undefined values. The function np.isnan() is used to check whether a value is np.nan.

#### Example:

```
import numpy as np
```

```
arr = np.array([1, 2, np.nan, 4])
```

```
# Checking for NaN
```

```
print(np.isnan(arr)) # Output: [False False True False]
```

```
# Replace NaN with a specific value (e.g., 0)
```

```
arr[np.isnan(arr)] = 0
```

```
print(arr) # Output: [1. 2. 0. 4.]
```

### b. np.nanmean()

**Definition:** np.nanmean() computes the mean of an array while ignoring np.nan values.

**Example:**

```
import numpy as np

arr = np.array([1, 2, np.nan, 4])

mean = np.nanmean(arr)
print(mean) # Output: 2.333333333333335
```

### **3. Performance Optimization**

Optimizing performance is crucial when working with large datasets. NumPy offers a few techniques to improve computational speed.

a. Using numba for Faster Computations

**Definition:** numba is a library that can significantly speed up NumPy operations by compiling Python code to machine code at runtime.

**Syntax:**

```
from numba import jit

@jit
def your_function(arr):
    # Perform operations on arr
    return result
```

**Example:**

```
import numpy as np
from numba import jit
```

@jit

```

def slow_function(arr):
    result = 0
    for i in range(len(arr)):
        result += arr[i]**2
    return result

arr = np.random.rand(1000000)
print(slow_function(arr))

```

Using the `@jit` decorator optimizes the function for faster execution.

### b. Vectorization vs Loops

**Definition:** Vectorization refers to applying operations to entire arrays at once, rather than iterating through them element by element (which can be slow). NumPy allows vectorized operations, making use of efficient C implementations.

#### Example:

- **Using loops:**

```

arr = np.random.rand(1000000)
result = 0
for i in range(len(arr)):
    result += arr[i]**2
print(result)

```

- **Using vectorization:**

```

arr = np.random.rand(1000000)
result = np.sum(arr**2)
print(result)

```

Vectorization eliminates the need for explicit loops and leverages NumPy's optimized functions, making it much faster.

## Conclusion

By loading and saving real-world data, handling missing values, and optimizing performance with tools like numba and vectorization, you can effectively work with large datasets and perform complex mathematical operations efficiently. Try implementing the examples and see the difference in performance when working with large-scale data.

## Mini Project 1: Data Analysis with Missing Values

### Goal:

Analyze a dataset, handle missing values, and perform basic statistics (mean, median, etc.) while optimizing performance.

### Step 1: Load the Data

1. **Create or download a data file** (e.g., a .csv or .txt file) that contains missing values. For example, a file data.txt might look like:

```
10, 20, 30  
40, NaN, 60  
70, 80, 90  
100, 110, 120
```

2. **Load the data using np.genfromtxt():**

- a. This will handle missing values (NaN) and read the data into a NumPy array.

```
import numpy as np
```

```
# Load data from a file
data = np.genfromtxt('data.txt', delimiter=',', filling_values=0)

# Print the loaded data
print(data)
```

## Step 2: Handle Missing Values

### 3. Identify missing values (NaN) using np.isnan():

- a. Check which elements are NaN in the dataset.

```
# Identify NaN values in the array
print(np.isnan(data)) # Output: [[False False False], [False True False], [False
False False], [False False False]]
```

### 4. Fill the NaN values with the mean of the column using np.nanmean():

- a. Compute the mean while ignoring NaN values and fill the missing data with these values.

```
# Fill NaN values with column-wise mean
for i in range(data.shape[1]):
    column_mean = np.nanmean(data[:, i])
    data[np.isnan(data[:, i]), i] = column_mean
```

```
# Print the data after handling missing values
print(data)
```

## Step 3: Perform Basic Statistics

### 5. Calculate statistics like mean, median, and standard deviation using NumPy functions:

```
# Mean, median, and standard deviation of the array
mean = np.mean(data)
```

```
median = np.median(data)
std_dev = np.std(data)

# Print results
print(f'Mean: {mean}, Median: {median}, Standard Deviation: {std_dev}')
```

#### **Step 4: Save the Processed Data**

6. **Save the processed data** back into a new file using `np.savetxt()`:

```
# Save the processed data to a new file
np.savetxt('processed_data.txt', data, delimiter=',', fmt='%.2f')
```

#### **Step 5: Performance Optimization with numba**

7. **Optimize the calculations using numba:**

- a. Here, we'll create a function to perform some calculations and use numba to speed up the loop-based operations.

```
from numba import jit
```

```
# Function to compute sum of squares (slow version using loop)
```

```
@jit
def sum_of_squares(arr):
    result = 0
    for i in range(len(arr)):
        result += arr[i] ** 2
    return result
```

```
# Test with a large array
```

```
large_data = np.random.rand(1000000)
```

```
# Measure execution time with numba optimization
```

```
import time
```

```

start = time.time()
print(sum_of_squares(large_data)) # Optimized version
print("Execution time with numba:", time.time() - start)

```

## Step 6: Vectorization vs Loops

### 8. Compare performance of vectorized vs loop-based operations:

- a. Vectorized operations are faster than loops because NumPy performs these operations in C.

```

# Vectorized version
start = time.time()
print(np.sum(large_data**2)) # Vectorized version
print("Execution time with vectorization:", time.time() - start)

```

## Mini Project 2: Optimizing Data Filtering with Real-World Dataset

### Goal:

Load a large dataset, filter specific rows based on conditions, and optimize performance using vectorization and numba.

### Step 1: Load the Data

1. Download or create a dataset (e.g., sales\_data.txt) containing sales data with columns such as Date, Product, Quantity, Price. Sample data:

2021-01-01, A, 10, 5.0  
 2021-01-02, B, NaN, 7.5  
 2021-01-03, A, 20, 5.5  
 2021-01-04, C, 15, 6.0

2. Load the data using np.genfromtxt():

```
import numpy as np

# Load sales data
data = np.genfromtxt('sales_data.txt', delimiter=',', dtype=None, encoding='utf-8',
names=True)

# Print the data
print(data)
```

### Step 2: Filter Data Based on Condition

3. **Filter out rows where Quantity is missing (i.e., NaN) using boolean masking:**

```
# Filter out rows where Quantity is NaN
filtered_data = data[~np.isnan(data['Quantity'])]
```

```
# Print filtered data
print(filtered_data)
```

### Step 3: Calculate Total Sales

4. **Calculate the total sales (i.e., Quantity \* Price) and store it in a new column:**

```
# Calculate total sales for each product
filtered_data = np.lib.recfunctions.append_fields(filtered_data, 'Total_Sales',
filtered_data['Quantity'] * filtered_data['Price'])

# Print the updated data with total sales
print(filtered_data)
```

#### **Step 4: Vectorization for Faster Operations**

5. **Use vectorized operations** to filter data for a specific condition (e.g., filter products with total sales greater than 100):

```
# Vectorized filtering for products with total sales > 100
high_sales = filtered_data[filtered_data['Total_Sales'] > 100]

# Print the result
print(high_sales)
```

#### **Step 5: Performance Optimization with numba**

6. **Optimize sales calculation using numba:**

```
from numba import jit
```

```
# Slow version of total sales calculation using loop
@jit
def calculate_sales_loop(data):
    total_sales = 0
    for i in range(len(data)):
        total_sales += data[i]['Quantity'] * data[i]['Price']
    return total_sales
```

```
# Measure execution time with numba optimization
start = time.time()
print(calculate_sales_loop(filtered_data)) # Optimized function
print("Execution time with numba:", time.time() - start)
```

#### **Step 6: Save Filtered Data**

7. **Save the filtered data** to a new text file using np.savetxt():

```
# Save the filtered data with total sales  
np.savetxt('filtered_sales_data.txt', filtered_data, delimiter=',', fmt='%s')
```

## Conclusion

In these two mini projects, you learned how to:

- Load and save real-world data using various NumPy functions.
- Handle missing values and perform basic statistical calculations.
- Optimize performance using vectorization and numba.
- Filter, analyze, and save the processed data efficiently.

You can scale these projects to work with larger datasets or integrate more complex analysis techniques as needed.

## Day 46 tasks

### 1. Loading Data with np.loadtxt()

- Load a text file that contains a dataset (e.g., CSV format) using np.loadtxt(). Ensure the data is read correctly into a NumPy array.

### 2. Loading Data with np.genfromtxt()

- Load a dataset with missing values from a CSV file using np.genfromtxt(). Handle missing values by setting the filling\_values argument to a specific value (e.g., 0 or np.nan).

### 3. Saving Data with np.savetxt()

- Save a NumPy array to a text file in a CSV format using np.savetxt(). Ensure that the array is saved with a specified delimiter (e.g., comma).

#### **4. Saving Data with np.save()**

- Save a NumPy array to a binary .npy file using np.save(). Understand the advantages of using this method over text file saving.

#### **5. Loading Data with np.load()**

- Load a previously saved .npy file back into memory using np.load(). Verify that the data is loaded correctly and matches the original dataset.

#### **6. Handling Missing Values with np.isnan()**

- Given a dataset with NaN values, use np.isnan() to find the positions of missing values and handle them (e.g., replacing them with the mean of the column).

#### **7. Handling Missing Values with np.nanmean()**

- Given an array with missing values, use np.nanmean() to calculate the mean while ignoring NaN values. Apply this technique to a dataset with missing entries.

#### **8. Performance Optimization with numba**

- Use the @jit decorator from numba to speed up a custom function that performs mathematical computations on large arrays. Measure and compare the execution time before and after optimization.

#### **9. Vectorized Operations vs. Loops**

- Create a NumPy array and perform an operation (e.g., element-wise addition) first using loops and then using vectorized operations. Compare the performance of both approaches for large datasets.

## 10. Optimizing Performance with numba for Array Manipulation

- Optimize a function that calculates the cumulative sum of elements in a large NumPy array using numba. Measure the speed-up compared to a regular Python loop.

## 11. Handling Multiple Missing Values in a Dataset

- Given a dataset with several columns, handle missing values in different ways, such as replacing missing values in numeric columns with column mean and filling categorical columns with a default value.

## 12. Performing Conditional Selection with np.where()

- Given a NumPy array, use np.where() to perform conditional selection. For example, select values greater than a certain threshold or assign a new value based on a condition.

## 13. Saving and Loading Large Arrays Efficiently

- Create a large array and save it using np.save(). Then, load it using np.load(). Ensure that the performance and memory usage are optimized for large arrays.

## Mini Project 1: Data Preprocessing and Optimization

**Objective:** You are given a dataset in CSV format containing numerical values and missing values (NaNs). Your task is to load the data, handle missing values, and optimize the computation of column-wise statistics using both loops and vectorization.

**Tasks:**

1. Load the dataset: Use `np.loadtxt()` or `np.genfromtxt()` to load the data from a CSV file.
2. Handle Missing Values: Identify missing values using `np.isnan()` and replace them with the column-wise mean (using `np.nanmean()` to ignore NaNs).
3. Calculate Statistics:
  - a. Calculate the mean, median, standard deviation, and variance of each column, both using loops and using vectorized operations.
4. Performance Comparison: Measure and compare the execution time of the loop-based and vectorized approaches for large datasets.
5. Save the Cleaned Data: After handling missing values, save the cleaned data using `np.savetxt()` or `np.save()`.

## **Mini Project 2: Data Loading and Fast Computation of Matrix Operations**

**Objective:** You are given two large matrices saved as .txt or .npy files. Your task is to load the matrices, perform matrix operations, and optimize performance using numba.

**Tasks:**

1. Load Data: Load the two matrices from .npy files using `np.load()`.
2. Matrix Operations: Perform matrix multiplication, element-wise addition, and find the determinant of one of the matrices using `np.matmul()`, `np.add()`, and `np.linalg.det()`.
3. Optimization with numba: Optimize the matrix multiplication function using numba's `@jit` decorator to speed up the computations.
4. Compare Performance: Measure the execution time of the matrix operations before and after applying numba optimization.
5. Save Results: Save the results of the matrix operations into a new .npy file.

## Bonus: Mini Projects Using NumPy

- Generate Random Data & Perform Statistical Analysis
- Implement a Simple Linear Regression Model using NumPy
- Create a NumPy-based Image Processing System
- Simulate Dice Rolls and Analyze Outcomes

# Pandas

## Day 47

### Mini Project 1: Employee Data Analysis using Pandas

#### Overview

In this project, we will create an Employee Data Analysis System using Pandas.

This will involve:

- ✓ Creating a Pandas DataFrame using employee details
- ✓ Performing data analysis – retrieving employee statistics
- ✓ Reading and Writing CSV files

#### Step 1: Install and Import Pandas

##### Install Pandas

If Pandas is not installed, install it using:

```
pip install pandas
```

## Import Pandas

```
import pandas as pd
```

## Step 2: Create an Employee DataFrame

### Using a Dictionary

```
# Employee Data
employee_data = {
    'Employee_ID': [101, 102, 103, 104, 105],
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Department': ['HR', 'IT', 'Finance', 'IT', 'HR'],
    'Salary': [50000, 70000, 60000, 80000, 55000],
    'Experience (Years)': [5, 7, 6, 9, 4]
}
```

```
# Creating DataFrame
df = pd.DataFrame(employee_data)
```

```
# Display DataFrame
print(df)
```

### Output

	Employee_ID	Name	Department	Salary	Experience (Years)
0	101	Alice	HR	50000	5
1	102	Bob	IT	70000	7
2	103	Charlie	Finance	60000	6
3	104	David	IT	80000	9
4	105	Eve	HR	55000	4

### **Step 3: Check Data Properties**

#### **Display First Few Rows**

```
print(df.head(3)) # First 3 rows
```

#### **Display Last Few Rows**

```
print(df.tail(2)) # Last 2 rows
```

#### **Get Shape of the Data**

```
print(df.shape) # (rows, columns)
```

#### **Get General Information**

```
print(df.info())
```

#### **Get Summary Statistics**

```
print(df.describe()) # Shows mean, min, max, std, etc.
```

### **Step 4: Save Data to a CSV File**

```
df.to_csv("employee_data.csv", index=False) # Save without index  
print("CSV file saved successfully!")
```

### **Step 5: Read Data from CSV**

```
df_new = pd.read_csv("employee_data.csv")  
print(df_new.head()) # Display first few rows
```

## Summary

- ✓ Installed and imported Pandas
- ✓ Created an Employee DataFrame
- ✓ Performed data analysis using .head(), .tail(), .info(), .describe()
- ✓ Saved and read data from a CSV file

## Mini Project 2: Sales Data Analysis using Pandas

### Overview

In this project, we will create a Sales Data Analysis System using Pandas: ✓  
Creating a Pandas DataFrame using sales records

- ✓ Performing data analysis – retrieving statistics and insights
- ✓ Reading data from CSV

### Step 1: Install and Import Pandas

```
import pandas as pd
```

### Step 2: Create a Sales DataFrame

#### Using a List of Dictionaries

```
# Sales Data
sales_data = [
    {'Product': 'Laptop', 'Category': 'Electronics', 'Units Sold': 10, 'Price per Unit': 50000},
    {'Product': 'Mobile', 'Category': 'Electronics', 'Units Sold': 25, 'Price per Unit': 20000},
```

```

{'Product': 'Tablet', 'Category': 'Electronics', 'Units Sold': 15, 'Price per Unit': 30000},
{'Product': 'Headphones', 'Category': 'Accessories', 'Units Sold': 50, 'Price per Unit': 2000},
{'Product': 'Mouse', 'Category': 'Accessories', 'Units Sold': 40, 'Price per Unit': 1500}
]

# Creating DataFrame
df_sales = pd.DataFrame(sales_data)

# Display DataFrame
print(df_sales)

```

**Output**

	Product	Category	Units Sold	Price per Unit
0	Laptop	Electronics	10	50000
1	Mobile	Electronics	25	20000
2	Tablet	Electronics	15	30000
3	Headphones	Accessories	50	2000
4	Mouse	Accessories	40	1500

**Step 3: Add a New Column (Total Sales)**

```

df_sales['Total Sales'] = df_sales['Units Sold'] * df_sales['Price per Unit']
print(df_sales)

```

**Output**

	Product	Category	Units Sold	Price per Unit	Total Sales
0	Laptop	Electronics	10	50000	500000

1	Mobile Electronics	25	20000	500000
2	Tablet Electronics	15	30000	450000
3	Headphones Accessories	50	2000	100000
4	Mouse Accessories	40	1500	60000

## Step 4: Check Data Properties

### Get First Few Rows

```
print(df_sales.head())
```

### Get Data Info

```
print(df_sales.info())
```

### Get Summary Statistics

```
print(df_sales.describe())
```

## Step 5: Save Sales Data to CSV

```
df_sales.to_csv("sales_data.csv", index=False)
print("Sales data saved to CSV file successfully!")
```

## Step 6: Read Data from CSV

```
df_read = pd.read_csv("sales_data.csv")
print(df_read.head()) # Display first few rows
```

## Summary

- ✓ Created a Sales DataFrame
- ✓ Added a Total Sales column

- ✓ Performed data analysis using .head(), .info(), .describe()
- ✓ Saved and loaded data from a CSV file

## Real-life mini-projects

### Mini Project 1: Employee Salary Data Analysis

**Objective:** Analyze employee salary data and extract meaningful insights.

#### Step 1: Install & Import Pandas

```
# Install pandas if not already installed
```

```
# pip install pandas
```

```
# Import Pandas
```

```
import pandas as pd
```

#### Step 2: Create Employee Salary DataFrame

We will create a DataFrame from a dictionary.

```
# Creating Employee Salary Data
```

```
employee_data = {
```

```
    "Employee_ID": [101, 102, 103, 104, 105],
```

```
    "Name": ["John", "Emma", "Alex", "Sophie", "David"],
```

```
    "Department": ["HR", "IT", "Finance", "IT", "HR"],
```

```
    "Salary": [60000, 80000, 75000, 85000, 65000],
```

```
    "Experience (Years)": [5, 7, 6, 8, 4]
```

```
}
```

```
# Convert dictionary to DataFrame
```

```
df = pd.DataFrame(employee_data)
```

**Step 3: Check Data Properties**

```
# Display the first few rows
```

```
print(df.head())
```

```
# Check DataFrame shape
```

```
print("Shape:", df.shape)
```

```
# Get data information
```

```
print(df.info())
```

```
# Get summary statistics
```

```
print(df.describe())
```

**Step 4: Access and Modify Columns**

```
# Select specific columns
```

```
print(df[["Name", "Salary"]])
```

```
# Increase salary by 10%
```

```
df["Salary"] = df["Salary"] * 1.10
```

```
# Rename the Salary column
```

```
df.rename(columns={"Salary": "Updated Salary"}, inplace=True)
```

**Step 5: Filter Employees Based on Experience**

```
# Employees with more than 5 years of experience
```

```
experienced_employees = df[df["Experience (Years)"] > 5]
```

```
print(experienced_employees)
```

**Step 6: Save Cleaned Data to CSV**

```
df.to_csv("employee_salary_data.csv", index=False)
```

**Mini Project 2: Student Marks Analysis**

**Objective:** Perform analysis on student exam scores and generate insights.

**Step 1: Install & Import Pandas**

```
import pandas as pd
```

**Step 2: Create a DataFrame from Lists**

```
# Creating Student Marks Data
student_data = {
    "Student_ID": [1, 2, 3, 4, 5],
    "Name": ["Alice", "Bob", "Charlie", "Diana", "Ethan"],
    "Maths": [85, 78, 92, 74, 88],
    "Science": [89, 82, 95, 80, 85],
    "English": [90, 75, 88, 70, 78]
}
```

```
# Convert to DataFrame
df = pd.DataFrame(student_data)
```

**Step 3: Check Data Properties**

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

```
# Check DataFrame shape
print("Shape:", df.shape)
```

```
# Get data info
print(df.info())

# Get summary statistics
print(df.describe())
```

**Step 4: Calculate Total & Average Marks**

```
# Add Total Marks column
df["Total Marks"] = df["Maths"] + df["Science"] + df["English"]

# Add Average Marks column
df["Average Marks"] = df["Total Marks"] / 3
```

**Step 5: Identify Top Performing Students**

```
# Find students with an average score above 85
top_students = df[df["Average Marks"] > 85]
print(top_students)
```

**Step 6: Save Data to CSV**

```
df.to_csv("student_marks_data.csv", index=False)
```

**Mini Project: Customer Purchase Data Analysis using Pandas****Overview**

This project involves analyzing customer purchase data using Pandas. The dataset contains customer details, product information, and transaction records. We will perform various tasks to clean, explore, and analyze the data.

## Day 47 Tasks

### 1. Install and import Pandas

- Install Pandas using pip install pandas.
- Import Pandas in Python using import pandas as pd.

### 2. Create a Pandas DataFrame

- Generate a dataset containing customer purchases using lists and dictionaries.

### 3. Load data from an external CSV file

- Read a customer purchase dataset from a CSV file using pd.read\_csv().

### 4. Check the first and last few rows of the DataFrame

- Use .head() and .tail() to examine the first and last few records.

### 5. Retrieve DataFrame properties

- Display the number of rows and columns using .shape.
- Get metadata about columns using .info().

### 6. Get summary statistics

- Use .describe() to compute statistics like mean, min, max, and standard deviation.

### 7. Access and modify specific columns

- Select specific columns such as 'Customer Name', 'Product', and 'Amount'.
- Modify column values using Pandas operations.

### 8. Filter customer data based on conditions

- Retrieve records where the purchase amount exceeds a certain threshold.

## 9. Sort the dataset

- Sort data based on purchase amount in descending order.

## 10. Handle missing values

- Detect missing values using `.isnull()`.
- Fill or remove missing values as required.

## 11. Group and aggregate data

- Find the total amount spent by each customer using `.groupby()`.

## 12. Add a new calculated column

- Create a column for discount (e.g., 10% discount for purchases above a threshold).

## 13. Save cleaned data to a new CSV file

- Store the cleaned and processed dataset using `.to_csv()`.

## Mini Project 1: Sales Data Analysis

### Objective:

Analyze sales data from an e-commerce business to extract meaningful insights.

### Requirements:

1. Create a Sales DataFrame from a CSV file containing product sales.
2. Display the first and last five rows of the dataset.
3. Extract column names and data types of each column.
4. Check the shape, missing values, and summary statistics.

5. Filter out sales above a certain amount (e.g., ₹10,000).
6. Add a new column "Profit Margin" based on revenue and cost price.
7. Find the top-selling product based on total revenue.
8. Group sales by month and calculate total sales per month.
9. Sort data by highest revenue and display the top 5 entries.
10. Find the average revenue per product category.
11. Identify underperforming products (sales below a threshold).
12. Save the cleaned DataFrame into a new CSV file.
13. Visualize key sales trends using matplotlib/seaborn.

## Mini Project 2: COVID-19 Data Analysis

### Objective:

Analyze COVID-19 data to track cases, recoveries, and deaths across different countries.

### Requirements:

1. Import COVID-19 data from a CSV file or API into a Pandas DataFrame.
2. Display basic information about the dataset (column names, types, etc.).
3. Check for missing values and handle them appropriately.
4. Extract the top 10 countries with the highest total cases.
5. Calculate active cases (total cases - recoveries - deaths) for each country.
6. Find the country with the highest death rate (deaths/total cases).
7. Group data by continent and calculate total cases per continent.
8. Find the average recovery rate across all countries.
9. Identify countries with a recovery rate above 90%.
10. Sort the dataset by new cases per day in descending order.
11. Save the cleaned data into a new CSV file.
12. Create line plots or bar charts to visualize case trends over time.
13. Predict future cases using basic linear regression (optional).

# Day 48

## Indexing, Selecting & Filtering Data in Pandas

When working with large datasets, it's important to extract specific parts of the data efficiently. Pandas provides multiple methods for indexing, selecting, and filtering data in DataFrames.

### 1. Indexing and Slicing in DataFrames (.loc[], .iloc[])

#### Definition

- **Indexing:** Refers to selecting specific rows or columns from a DataFrame using labels or positions.
- **Slicing:** Allows extracting a subset of rows and columns from a DataFrame.

#### Syntax

```
df.loc[row_label, column_label] # Selects data using labels (index-based)
df.iloc[row_position, column_position] # Selects data using integer positions
(zero-based)
```

#### Example: Using .loc[] and .iloc[]

##### Step 1: Create a Sample DataFrame

```
import pandas as pd
```

```
# Creating a sample DataFrame
data = {
    "Employee": ["John", "Emma", "Alex", "Sophia", "Michael"],
    "Department": ["HR", "IT", "Finance", "Marketing", "IT"],
    "Salary": [50000, 70000, 65000, 60000, 72000]
}
```

```
df = pd.DataFrame(data)
print(df)
```

**Output:**

```
Employee Department Salary
0 John HR 50000
1 Emma IT 70000
2 Alex Finance 65000
3 Sophia Marketing 60000
4 Michael IT 72000
```

**Using .loc[] (Label-based selection)****Step 2: Select specific rows and columns**

```
# Select employee names and their departments
print(df.loc[:, ["Employee", "Department"]])
```

**Output:**

```
Employee Department
0 John HR
1 Emma IT
2 Alex Finance
3 Sophia Marketing
4 Michael IT
```

```
# Select a specific row by index label (default index)
print(df.loc[2])
```

**Output:**

```
Employee Alex
Department Finance
Salary 65000
Name: 2, dtype: object
```

### Using .iloc[] (Position-based selection)

#### Step 3: Select specific rows and columns by index position

```
# Select the first two rows
```

```
print(df.iloc[0:2])
```

#### Output:

```
Employee Department Salary
```

```
0 John HR 50000
```

```
1 Emma IT 70000
```

```
# Select the second row and first column
```

```
print(df.iloc[1, 0]) # Output: Emma
```

## 2. Selecting Specific Columns and Rows

### Selecting Specific Columns

```
print(df["Employee"]) # Selecting a single column
```

#### Output:

```
0 John
```

```
1 Emma
```

```
2 Alex
```

```
3 Sophia
```

```
4 Michael
```

```
Name: Employee, dtype: object
```

```
# Selecting multiple columns
```

```
print(df[["Employee", "Salary"]])
```

#### Output:

```
Employee Salary
```

```
0 John 50000
```

```
1 Emma 70000
2 Alex 65000
3 Sophia 60000
4 Michael 72000
```

### 3. Boolean Indexing & Filtering

#### Definition

Boolean indexing allows filtering based on conditions.

#### Syntax

```
df[condition]
```

#### Example: Filtering Employees with Salary > 60,000

```
high_salary = df[df["Salary"] > 60000]
print(high_salary)
```

#### Output:

```
Employee Department Salary
1 Emma      IT    70000
2 Alex       Finance 65000
4 Michael    IT    72000
```

### 4. Using isin() and between() for Filtering

#### Using isin() (Checking for Multiple Values)

```
# Filter employees working in IT or HR
filter_dept = df[df["Department"].isin(["IT", "HR"])]
print(filter_dept)
```

**Output:**

```
Employee Department Salary
0 John      HR 50000
1 Emma      IT 70000
4 Michael   IT 72000
```

**Using between() (Filtering Numerical Ranges)**

```
# Employees with salary between 55,000 and 70,000
salary_range = df[df["Salary"].between(55000, 70000)]
print(salary_range)
```

**Output:**

```
Employee Department Salary
2 Alex      Finance 65000
3 Sophia    Marketing 60000
```

**5. Changing Index (set\_index(), reset\_index())****Setting a New Index**

```
df_new = df.set_index("Employee")
print(df_new)
```

**Output:**

```
Department Salary
Employee
John      HR 50000
Emma     IT 70000
Alex     Finance 65000
Sophia   Marketing 60000
Michael  IT 72000
```

### Resetting the Index

```
df_reset = df_new.reset_index()
print(df_reset)
```

### Output (Restored Original Format):

Employee Department Salary

```
0 John HR 50000
1 Emma IT 70000
2 Alex Finance 65000
3 Sophia Marketing 60000
4 Michael IT 72000
```

## Summary

Concept	Method/Function	Example
Indexing by label	.loc[]	df.loc[1, "Employee"]
Indexing by position	.iloc[]	df.iloc[1, 0]
Selecting Columns	df["column_name"], df[["col1", "col2"]]	df["Salary"]
Filtering by Condition	df[condition]	df[df["Salary"] > 60000]
Filtering using isin()	.isin()	df[df["Department"].isin(["IT", "HR"])]
Filtering using between()	.between()	df[df["Salary"].between(55000, 70000)]
Changing Index	set_index("column")	df.set_index("Employee")
Resetting Index	reset_index()	df.reset_index()

## Real-Life Example

Scenario: An HR department wants to analyze employee salaries and filter high-paying jobs.

### Step-by-Step Implementation

```
import pandas as pd
```

```
# Creating the employee dataset
```

```
data = {  
    "Employee": ["John", "Emma", "Alex", "Sophia", "Michael"],  
    "Department": ["HR", "IT", "Finance", "Marketing", "IT"],  
    "Salary": [50000, 70000, 65000, 60000, 72000]  
}
```

```
df = pd.DataFrame(data)
```

```
# Selecting Employees with Salary > 60,000
```

```
high_salary_employees = df[df["Salary"] > 60000]
```

```
# Setting Employee as index
```

```
df_indexed = df.set_index("Employee")
```

```
# Resetting index to original
```

```
df_reset = df_indexed.reset_index()
```

```
# Displaying results
```

```
print(high_salary_employees)
```

```
print(df_indexed)
```

```
print(df_reset)
```

## Key Takeaways

- ✓ .loc[] and .iloc[] are powerful tools for selecting data by labels or positions.
- ✓ Boolean indexing enables efficient filtering of data.
- ✓ isin() and between() are useful for multi-value filtering and range filtering.
- ✓ Index manipulation (set\_index(), reset\_index()) allows restructuring of DataFrames.

## Mini Project 1: Sales Data Analysis

### Problem Statement

A company wants to analyze its sales data to:

- Filter high-selling products.
- Find sales from specific locations.
- Extract details of products sold within a given price range.

### Dataset Sample

The dataset contains the following columns:

- Product: Name of the product.
- Category: Product category (Electronics, Furniture, etc.).
- Price: Product price in USD.
- Quantity Sold: Number of units sold.
- City: City where the product was sold.

### Step 1: Install & Import Pandas

```
import pandas as pd
```

**Step 2: Create a Sample DataFrame**

```
# Creating a sales dataset
data = {
    "Product": ["Laptop", "Smartphone", "Table", "Headphones", "Chair",
    "Monitor"],
    "Category": ["Electronics", "Electronics", "Furniture", "Electronics", "Furniture",
    "Electronics"],
    "Price": [1200, 800, 300, 150, 200, 400],
    "Quantity Sold": [50, 100, 20, 70, 30, 40],
    "City": ["New York", "Los Angeles", "Chicago", "New York", "Chicago", "Los
    Angeles"]
}

df = pd.DataFrame(data)

print(df)
```

**Output:**

	Product	Category	Price	Quantity Sold	City
0	Laptop	Electronics	1200	50	New York
1	Smartphone	Electronics	800	100	Los Angeles
2	Table	Furniture	300	20	Chicago
3	Headphones	Electronics	150	70	New York
4	Chair	Furniture	200	30	Chicago
5	Monitor	Electronics	400	40	Los Angeles

**Step 3: Select Specific Columns**

```
# Selecting only Product and Price columns
df_selected = df[["Product", "Price"]]
print(df_selected)
```

**Output:**

	Product	Price
0	Laptop	1200
1	Smartphone	800
2	Table	300
3	Headphones	150
4	Chair	200
5	Monitor	400

**Step 4: Select Specific Rows**

```
# Selecting the first 3 rows
df_first_3 = df.iloc[:3]
print(df_first_3)
```

**Output:**

	Product	Category	Price	Quantity Sold	City
0	Laptop	Electronics	1200	50	New York
1	Smartphone	Electronics	800	100	Los Angeles
2	Table	Furniture	300	20	Chicago

**Step 5: Boolean Indexing (Products with Price > 500)**

```
# Filtering products with price greater than $500
expensive_products = df[df["Price"] > 500]
print(expensive_products)
```

**Output:**

	Product	Category	Price	Quantity Sold	City
0	Laptop	Electronics	1200	50	New York
1	Smartphone	Electronics	800	100	Los Angeles

**Step 6: Using isin() (Filter Sales from Specific Cities)**

```
# Filtering products sold in New York or Chicago
filtered_cities = df[df["City"].isin(["New York", "Chicago"])]
print(filtered_cities)
```

**Output:**

	Product	Category	Price	Quantity Sold	City
0	Laptop	Electronics	1200	50	New York
2	Table	Furniture	300	20	Chicago
3	Headphones	Electronics	150	70	New York
4	Chair	Furniture	200	30	Chicago

**Step 7: Using between() (Filter Products within Price Range)**

```
# Filtering products priced between $200 and $800
mid_price_products = df[df["Price"].between(200, 800)]
print(mid_price_products)
```

**Output:**

	Product	Category	Price	Quantity Sold	City
1	Smartphone	Electronics	800	100	Los Angeles
2	Table	Furniture	300	20	Chicago
4	Chair	Furniture	200	30	Chicago
5	Monitor	Electronics	400	40	Los Angeles

**Step 8: Changing Index to "Product"**

```
df_indexed = df.set_index("Product")
print(df_indexed)
```

**Output:**

Product	Category	Price	Quantity Sold	City
Laptop	Electronics	1200	50	New York
Smartphone	Electronics	800	100	Los Angeles
Table	Furniture	300	20	Chicago
Headphones	Electronics	150	70	New York
Chair	Furniture	200	30	Chicago
Monitor	Electronics	400	40	Los Angeles

**Step 9: Reset Index**

```
df_reset = df_indexed.reset_index()
print(df_reset)
```

**Mini Project 2: Student Data Analysis****Problem Statement**

A university wants to analyze student performance. You need to:

- Extract details of students with grades above 80.
- Find students from specific courses.
- Select students based on age range.

**Dataset Sample**

- Student: Name of the student.
- Course: Course enrolled (Computer Science, Physics, etc.).
- Age: Age of the student.
- Grade: Percentage grade in the subject.
- City: City of residence.

**Step 1: Create a Sample DataFrame**

```
data = {  
    "Student": ["Alice", "Bob", "Charlie", "David", "Eva"],  
    "Course": ["CS", "Physics", "CS", "Math", "Physics"],  
    "Age": [20, 22, 21, 23, 20],  
    "Grade": [85, 78, 92, 88, 76],  
    "City": ["New York", "Los Angeles", "Chicago", "New York", "Chicago"]  
}  
  
df = pd.DataFrame(data)  
print(df)
```

**Step 2: Selecting Specific Columns**

```
df_selected = df[["Student", "Grade"]]  
print(df_selected)
```

**Step 3: Filtering Students with Grade > 80**

```
high_grade_students = df[df["Grade"] > 80]  
print(high_grade_students)
```

**Step 4: Filtering Students in Specific Courses**

```
cs_students = df[df["Course"].isin(["CS"])]  
print(cs_students)
```

**Step 5: Selecting Students Aged Between 20 and 22**

```
age_filtered_students = df[df["Age"].between(20, 22)]  
print(age_filtered_students)
```

**Step 6: Changing Index to Student Names**

```
df_indexed = df.set_index("Student")
print(df_indexed)
```

**Step 7: Resetting Index**

```
df_reset = df_indexed.reset_index()
print(df_reset)
```

**Conclusion**

Both projects show how Indexing, Selecting, and Filtering Data in Pandas is useful in real-world scenarios. You can now:

- ✓ Extract relevant data using .loc[], .iloc[]

- ✓ Filter using Boolean conditions, isin(), between()
- ✓ Change and reset indexes easily with set\_index() and reset\_index()

**Real-Life Mini Project: Employee Data Analysis****Problem Statement:**

A company wants to analyze its employee data to extract meaningful insights. You need to complete 13 tasks using Indexing, Selecting, and Filtering Data in Pandas.

**Dataset Sample (Columns to be used)**

- Employee\_ID: Unique identifier for each employee
- Name: Full name of the employee
- Department: Department of the employee (IT, HR, Finance, Sales, etc.)
- Age: Age of the employee
- Salary: Monthly salary in USD
- Experience: Number of years of work experience
- City: City where the employee is based

## Day 48 Tasks

1. Load the employee dataset into a Pandas DataFrame.
2. Display the first five and last five rows of the dataset.
3. Select and display only the "Name" and "Department" columns.
4. Retrieve employee details where "Department" is "IT".
5. Use indexing to select employees whose "Salary" is greater than \$5000.
6. Extract employees who have between 5 and 10 years of experience using `between()`.
7. Filter employees who are based in either "New York" or "Chicago" using `isin()`.
8. Use `.loc[]` to retrieve the details of an employee with a specific Employee\_ID.
9. Use `.iloc[]` to select the 2nd to 5th employees from the dataset.
10. Change the index of the DataFrame to "Employee\_ID" using `set_index()`.
11. Reset the index back to the default numerical index using `reset_index()`.
12. Find employees who are earning more than \$7000 and belong to the "Finance" department.
13. Retrieve employees whose age is greater than 30 and experience is less than 5 years using Boolean filtering.

## Expected Outcome

After completing these 13 tasks, you will gain hands-on experience in Indexing, Selecting, and Filtering Data in Pandas, which will help you handle real-world employee datasets efficiently.

## Real-Life Mini Project Requirements (Without Solutions)

### 1. Customer Purchase Data Analysis

#### Problem Statement:

An e-commerce company wants to analyze customer purchases to improve their marketing strategies. You need to extract meaningful insights from a dataset containing customer transactions.

#### Tasks to be performed:

- Load customer purchase data into a Pandas DataFrame.
- Display the first and last few rows of the dataset.
- Select specific columns such as "Customer\_ID", "Product\_Category", and "Purchase\_Amount".
- Retrieve data for customers who have purchased products from a specific category (e.g., "Electronics").
- Filter customers whose purchase amount is greater than \$500 using Boolean indexing.
- Extract transactions made between two specific dates using the between() function.
- Identify customers who made purchases in either "New York" or "San Francisco" using isin().
- Select specific rows using .loc[] and .iloc[].
- Change the index to "Customer\_ID" using set\_index().
- Reset the index back to the default numerical index using reset\_index().
- Find high-value customers who made purchases above \$1000 in the "Furniture" category.
- Retrieve customer details where the number of items purchased is between 3 and 10.

- Use Boolean filtering to find customers who made purchases above \$500 and belong to a specific region.

## 2. Healthcare Patient Records Filtering

### **Problem Statement:**

A hospital wants to analyze patient records to track high-risk patients based on their age, medical history, and health metrics. You need to filter patient records using indexing and selection techniques.

### **Tasks to be performed:**

- Load patient records into a Pandas DataFrame.
- Display the first and last few rows of the dataset.
- Select specific columns like "Patient\_ID", "Age", "Blood\_Pressure", and "Disease".
- Retrieve patient records where "Disease" is "Diabetes".
- Use Boolean indexing to filter patients whose "Age" is greater than 60.
- Extract records where "Blood\_Pressure" is between 120 and 140 using `between()`.
- Identify patients who live in either "Los Angeles" or "Houston" using `isin()`.
- Retrieve details of a specific patient using `.loc[]` based on their "Patient\_ID".
- Select a range of rows using `.iloc[]` for deeper analysis.
- Change the index to "Patient\_ID" using `set_index()`.
- Reset the index back to its default form using `reset_index()`.
- Find patients who have heart-related issues and are above 50 years old.
- Retrieve records of patients whose cholesterol levels are above 200 and have a family history of heart disease.

# Day 49

## Data Cleaning & Handling Missing Values in Pandas

### What is Data Cleaning? Why is it Important?

Data cleaning is the process of detecting and correcting (or removing) incorrect, inconsistent, or missing data from a dataset. It ensures that data is accurate, reliable, and ready for analysis.

For example, in real-world datasets (e.g., customer records, financial data, or medical records), missing values or incorrect data types can lead to misleading analysis.

### 1. Checking for Missing Data in Pandas

#### `isnull()` & `notnull()`

These functions help identify missing (NaN) values in a DataFrame.

#### Example: Checking Missing Values

```
import pandas as pd
```

```
# Sample dataset
data = {
    "Name": ["John", "Anna", "Mike", None, "Emma"],
    "Age": [25, None, 30, 22, 29],
    "Salary": [50000, 60000, None, 40000, 45000]
}
```

```
df = pd.DataFrame(data)
```

```
# Checking for missing values
```

```
print(df.isnull()) # Returns True for missing values  
  
# Checking for non-missing values  
print(df.notnull()) # Returns True for non-missing values  
  
# Count total missing values per column  
print(df.isnull().sum())
```

**Output:**

```
Name  Age  Salary  
0  False  False  False  
1  False  True  False  
2  False  False  True  
3  True  False  False  
4  False  False  False
```

```
Name    1  
Age    1  
Salary  1  
dtype: int64
```

## 2. Handling Missing Values

### Dropping Missing Values (dropna())

Used when we don't want to keep any rows with missing values.

#### Example: Removing Rows with Missing Values

```
df_cleaned = df.dropna()  
print(df_cleaned)
```

**Output:**

	Name	Age	Salary
0	John	25.0	50000.0
4	Emma	29.0	45000.0

**Note:** It removes rows with missing values completely.

**Filling Missing Values (fillna(), ffill(), bfill())**

Instead of dropping missing values, we can **fill them** with a default value.

**Example: Replacing Missing Values**

```
# Filling missing values with a specific value
df_filled = df.fillna({"Age": df["Age"].mean(), "Salary": df["Salary"].median()})
print(df_filled)
```

**Output (After Filling Missing Values)**

	Name	Age	Salary
0	John	25.0	50000.0
1	Anna	26.5	60000.0
2	Mike	30.0	45000.0
3	None	22.0	40000.0
4	Emma	29.0	45000.0

**Note:** df["Age"].mean() replaces missing values with the average age, and df["Salary"].median() fills missing salary with the median.

### **Forward Fill (ffill) and Backward Fill (bfill)**

```
df_ffill = df.ffill() # Forward Fill
df_bfill = df.bfill() # Backward Fill
```

#### **Explanation:**

- ffill() copies the previous value into the missing spot.
- bfill() copies the next value into the missing spot.

### **3. Handling Duplicates (drop\_duplicates())**

Duplicate values in a dataset can cause redundancy.

#### **Example: Removing Duplicate Rows**

```
df_duplicates = pd.DataFrame({
    "Name": ["John", "Anna", "John", "Mike", "Anna"],
    "Age": [25, 26, 25, 30, 26]
})
```

```
# Dropping duplicate rows
df_no_duplicates = df_duplicates.drop_duplicates()
print(df_no_duplicates)
```

#### **Output:**

	Name	Age
0	John	25
1	Anna	26
3	Mike	30

**Note:** The duplicate "John" and "Anna" rows have been removed.

## 4. Changing Data Types (`astype()`, `pd.to_datetime()`)

Data types may need to be changed for correct processing.

### Example: Converting Data Types

```
df["Age"] = df["Age"].astype("int") # Convert to integer
df["Salary"] = df["Salary"].astype("float") # Convert to float
print(df.dtypes)
```

### Output:

Name	object
Age	int64
Salary	float64
dtype:	object

**Note:** Converting Age to integer and Salary to float.

### Example: Converting to Date Format (`pd.to_datetime()`)

```
df_dates = pd.DataFrame({"Date": ["2024-03-01", "2024/03/02", "March 3, 2024"]})
df_dates["Date"] = pd.to_datetime(df_dates["Date"])
print(df_dates)
```

### Output:

Date
0 2024-03-01
1 2024-03-02
2 2024-03-03

**Note:** Converts different date formats to a standardized format.

## 5. Renaming Columns & Indexes (rename())

### Example: Renaming Columns

```
df_renamed = df.rename(columns={"Name": "Employee Name", "Salary": "Monthly Salary"})
print(df_renamed)
```

### Output:

	Employee Name	Age	Monthly Salary
0	John	25	50000.0
1	Anna	26	60000.0
2	Mike	30	45000.0
3	None	22	40000.0
4	Emma	29	45000.0

### Example: Renaming Index

```
df.index = ["A", "B", "C", "D", "E"]
print(df)
```

### Output:

	Name	Age	Salary
A	John	25.0	50000.0
B	Anna	26.0	60000.0
C	Mike	30.0	45000.0
D	None	22.0	40000.0
E	Emma	29.0	45000.0

**Note:** The index labels are changed from numbers to "A", "B", "C", etc..

## Summary Table

Task	Method
Check missing values	.isnull(), .notnull()
Remove missing values	.dropna()
Fill missing values	.fillna(), .ffill(), .bfill()
Remove duplicates	.drop_duplicates()
Change data type	.astype(), pd.to_datetime()
Rename columns & indexes	.rename()

## Real-World Scenario

Imagine you are working with hospital patient data, and some age and test results are missing.

- Use .isnull().sum() to check missing data.
- Replace missing age with the average using .fillna(df["Age"].mean()).
- Drop irrelevant rows using .dropna().
- Convert test date columns using pd.to\_datetime().
- Rename columns for clarity using .rename().

## Conclusion

Cleaning data is essential for accurate data analysis. Using Pandas, we can: ✓

Check for missing values

✓ Handle missing and duplicate data

✓ Convert data types and rename columns

Mastering these techniques will improve data quality and make analysis more effective!

## Project 1: Cleaning Customer Data for an E-commerce Business

### Project Goal:

An e-commerce company maintains a customer database that contains missing, duplicate, and inconsistent data. The goal is to clean this dataset to ensure accurate customer insights.

### Steps for Implementation:

#### Step 1: Import Required Libraries

```
import pandas as pd
```

```
import numpy as np
```

#### Step 2: Create a Sample Customer Dataset

```
data = {
```

```
    "Customer_ID": [101, 102, 103, 104, 105, 106, 107, 107],  
    "Name": ["Alice", "Bob", "Charlie", "David", None, "Eve", "Frank", "Frank"],  
    "Age": [25, 30, np.nan, 28, 32, 29, None, 40],  
    "Email": ["alice@mail.com", "bob@mail.com", "charlie@mail.com", None,  
              "eve@mail.com", "eve@mail.com", "frank@mail.com", "frank@mail.com"],  
    "Join_Date": ["2024-01-01", "2024-02-15", "2024-03-20", "2024-04-10", None,  
                 "2024-06-05", "2024-07-22", "2024-07-22"],  
    "Spending ($)": [500, np.nan, 700, 400, 600, 350, 800, 800]  
}
```

```
df = pd.DataFrame(data)  
print(df)
```

#### Step 3: Identify Missing Values

```
print(df.isnull().sum())
```

**Output:**

```
Customer_ID  0
Name         1
Age          2
Email        1
Join_Date    1
Spending ($) 1
dtype: int64
```

**Insight:** Several columns have missing values.

**Step 4: Drop Rows with Missing Email (Critical Data)**

```
df = df.dropna(subset=["Email"])
print(df)
```

**Explanation:** Since **email** is a critical identifier, we remove rows where it is missing.

**Step 5: Fill Missing Values in Age & Spending (\$)**

```
df["Age"].fillna(df["Age"].mean(), inplace=True) # Fill Age with average
df["Spending ($)"].fillna(df["Spending ($)"].median(), inplace=True) # Fill
Spending with median
print(df)
```

**Insight:**

- Missing Age values are filled with the mean.
- Missing Spending (\$) values are replaced with the median.

**Step 6: Convert Join\_Date to Date Format**

```
df["Join_Date"] = pd.to_datetime(df["Join_Date"])
print(df.dtypes)
```

**Insight:** Join\_Date is now in datetime64 format.

**Step 7: Remove Duplicate Customer Records**

```
df = df.drop_duplicates()
print(df)
```

**Insight:** Duplicate records of Customer\_ID 107 are removed.

**Step 8: Rename Columns for Clarity**

```
df = df.rename(columns={"Spending ($)": "Total_Spending", "Join_Date": "Registration_Date"})
print(df.head())
```

**Insight:** Column names are more meaningful.

**Final Cleaned Data:**

	Customer_ID	Name	Age	Email	Registration_Date	Total_Spending
0	101	Alice	25.0	<u>alice@mail.com</u>	2024-01-01	500.0
1	102	Bob	30.0	<u>bob@mail.com</u>	2024-02-15	575.0
2	103	Charlie	30.8	<u>charlie@mail.com</u>	2024-03-20	700.0
3	104	David	28.0	None	2024-04-10	400.0
5	106	Eve	29.0	<u>eve@mail.com</u>	2024-06-05	350.0
6	107	Frank	40.0	<u>frank@mail.com</u>	2024-07-22	800.0

## Project 2: Cleaning Employee Data for HR Analytics

### Project Goal:

An HR department has an employee dataset with missing, inconsistent, and duplicate data. The goal is to clean it for further analysis.

### Steps for Implementation:

#### Step 1: Create Sample Employee Data

```
data = {
    "Employee_ID": [1, 2, 3, 4, 5, 6, 7, 7],
    "Name": ["John", "Jane", "Mike", "Sara", "Chris", None, "David", "David"],
    "Department": ["IT", "HR", "IT", "Finance", None, "IT", "HR", "HR"],
    "Salary": [60000, 55000, None, 70000, 65000, 60000, None, 70000],
    "Joining_Date": ["2021-06-01", "2020-09-15", None, "2019-12-10", "2022-07-
20", "2023-03-01", "2021-01-01", "2021-01-01"]
}

df = pd.DataFrame(data)
print(df)
```

#### Step 2: Identify Missing Values

```
print(df.isnull().sum())
```

### Output:

```
Employee_ID    0
Name         1
Department    1
Salary        2
Joining_Date  1
dtype: int64
```

**Insight:** Some employee details are missing.

### Step 3: Fill Missing Department & Salary Values

```
df["Department"].fillna("Unknown", inplace=True)  
df["Salary"].fillna(df["Salary"].median(), inplace=True)  
print(df)
```

#### Insight:

- Department missing values are replaced with "Unknown".
- Salary missing values are replaced with the median salary.

### Step 4: Convert Joining\_Date to Date Format

```
df["Joining_Date"] = pd.to_datetime(df["Joining_Date"])  
print(df.dtypes)
```

#### Insight: Joining\_Date is now in the correct format.

### Step 5: Remove Duplicate Employee Records

```
df = df.drop_duplicates()  
print(df)
```

#### Insight: Duplicate records of Employee\_ID 7 are removed.

### Step 6: Rename Columns

```
df = df.rename(columns={"Salary": "Annual_Salary", "Joining_Date":  
"Start_Date"})  
print(df.head())
```

#### Insight: The columns now have more meaningful names.

**Final Cleaned Data:**

	Employee_ID	Name	Department	Annual_Salary	Start_Date
0	1	John	IT	60000.0	2021-06-01
1	2	Jane	HR	55000.0	2020-09-15
2	3	Mike	IT	62500.0	NaT
3	4	Sara	Finance	70000.0	2019-12-10
4	5	Chris	Unknown	65000.0	2022-07-20
5	6	None	IT	60000.0	2023-03-01
6	7	David	HR	62500.0	2021-01-01

Note:

- Employee names & departments are now complete.
- Salary values are filled.
- Duplicate records are removed.

**Summary of Key Functions Used**

Task	Function
Check missing data	.isnull(), .sum()
Drop missing values	.dropna()
Fill missing values	.fillna()
Convert data types	.astype(), pd.to_datetime()
Remove duplicates	.drop_duplicates()
Rename columns	.rename()

**Conclusion**

These two projects demonstrate real-world data cleaning techniques using Pandas. Cleaning data is essential for accurate analytics and business insights! 📈

## Real-Life Mini Project: Cleaning Sales Data for a Retail Business

### Project Goal:

A retail store has a sales dataset containing missing, duplicate, and inconsistent data. The goal is to clean the dataset to ensure accurate sales reporting and analysis.

### Day 49 Tasks

1. Load the dataset into a Pandas DataFrame from a CSV file.
2. Check for missing values in each column using `.isnull().sum()`.
3. Drop rows with critical missing values (e.g., missing `Product_ID` or `Sales_Amount`).
4. Fill missing values in `Price` with the average price of that product category.
5. Use forward fill (`ffill`) and backward fill (`bfill`) for missing values in `Customer_Location`.
6. Handle missing values in `Purchase_Date` by filling with the most frequent date.
7. Convert `Purchase_Date` to a datetime format using `pd.to_datetime()`.
8. Standardize column names by renaming (`rename()`) them for consistency.
9. Identify and remove duplicate sales transactions using `drop_duplicates()`.
10. Change the data type of `Product_ID` to string using `.astype(str)`.
11. Replace incorrect values in `Payment_Method` (e.g., fix typos like "Csh" → "Cash").
12. Sort data by `Purchase_Date` for better readability and analysis.
13. Save the cleaned dataset to a new CSV file for further analysis.

### Expected Outcome:

- A cleaned sales dataset free from missing, duplicate, and inconsistent data.
- Improved accuracy in reporting and decision-making for the retail business.

## 1. Real-Life Mini Project: Cleaning Customer Feedback Data for a Company

### Project Goal:

A company has collected customer feedback data, but the dataset contains missing values, duplicates, and inconsistent formats. The goal is to clean the dataset to prepare it for sentiment analysis and customer insights.

## 2. Real-Life Mini Project: Cleaning Employee Attendance Data for HR Analysis

### Project Goal:

An HR department has an employee attendance record with missing timestamps, duplicate entries, and incorrect data types. The goal is to clean the dataset to ensure accurate payroll and performance tracking.

# Day 50

## Data Transformation & Aggregation in Pandas

### Definition

Data transformation and aggregation are key processes in data analysis that help summarize, reshape, and manipulate large datasets to extract meaningful insights.

- **Data Transformation:** Applying functions, modifying values, and restructuring data.
- **Data Aggregation:** Grouping data and computing summary statistics (e.g., sum, mean, count).

- **Sorting & Pivoting:** Organizing and reshaping data for better readability.

## Syntax and Explanation

### 1. Applying Functions to Columns (`apply()`, `map()`, `lambda`)

Used to transform column values using a function.

#### Syntax:

```
df['column_name'].apply(function)
df['column_name'].map(function)
df['column_name'].apply(lambda x: operation)
```

#### Example: Applying a Function to Modify Values

**Scenario:** Suppose you have a dataset with product prices, and you want to apply a 10% discount to all products.

```
import pandas as pd

# Sample DataFrame
data = {'Product': ['Laptop', 'Phone', 'Tablet'],
        'Price': [1000, 500, 300]}

df = pd.DataFrame(data)

# Apply a discount using a lambda function
df['Discounted Price'] = df['Price'].apply(lambda x: x * 0.9)

print(df)
```

**Output:**

	Product	Price	Discounted Price
0	Laptop	1000	900.0
1	Phone	500	450.0
2	Tablet	300	270.0

**2. Grouping Data (groupby())**

Used to group data based on a specific column and apply aggregate functions.

**Syntax:**

```
df.groupby('column_name').aggregate_function()
```

**Example: Grouping Sales Data by Region**

**Scenario:** A company wants to know the total sales per region.

```
data = {'Region': ['East', 'West', 'East', 'West', 'East'],
        'Sales': [200, 150, 300, 250, 100]}
```

```
df = pd.DataFrame(data)
```

```
# Group by Region and calculate total sales
total_sales = df.groupby('Region')['Sales'].sum()
```

```
print(total_sales)
```

**Output:**

```
Region
East    600
West    400
Name: Sales, dtype: int64
```

### 3. Aggregation Functions (sum(), mean(), count(), min(), max())

Used to summarize grouped data.

#### Syntax:

```
df.groupby('column_name').agg({'col1': 'sum', 'col2': 'mean'})
```

#### Example: Aggregating Student Scores

**Scenario:** Calculate the average marks per subject.

```
data = {'Subject': ['Math', 'Science', 'Math', 'Science', 'Math'],
        'Marks': [80, 85, 90, 75, 95]}
```

```
df = pd.DataFrame(data)
```

```
# Calculate the average marks per subject
avg_marks = df.groupby('Subject')['Marks'].mean()

print(avg_marks)
```

#### Output:

```
Subject
Math    88.33
Science 80.00
Name: Marks, dtype: float64
```

### 4. Pivot Tables (pivot\_table())

Used to summarize data like an Excel Pivot Table.

**Syntax:**

```
df.pivot_table(values='column', index='column', columns='column',
aggfunc='function')
```

**Example: Sales Pivot Table**

**Scenario:** You want to see the total sales per product per region.

```
data = {'Region': ['East', 'West', 'East', 'West', 'East'],
        'Product': ['Laptop', 'Phone', 'Tablet', 'Laptop', 'Phone'],
        'Sales': [1000, 700, 300, 1200, 800]}
```

```
df = pd.DataFrame(data)
```

```
# Create a pivot table
```

```
pivot = df.pivot_table(values='Sales', index='Region', columns='Product',
aggfunc='sum')
```

```
print(pivot)
```

**Output:**

Region	Product	Laptop	Phone	Tablet
East		1000.0	800.0	300.0
West		1200.0	700.0	NaN

**5. Sorting Data (sort\_values(), sort\_index())**

Used to arrange data in ascending or descending order.

**Syntax:**

```
df.sort_values(by='column', ascending=True/False)
df.sort_index()
```

### Example: Sorting Products by Price

**Scenario:** Sort products based on price from highest to lowest.

```
data = {'Product': ['Laptop', 'Phone', 'Tablet'],
        'Price': [1000, 500, 300]}

df = pd.DataFrame(data)

# Sort by price in descending order
df_sorted = df.sort_values(by='Price', ascending=False)

print(df_sorted)
```

#### Output:

```
Product  Price
0 Laptop  1000
1 Phone   500
2 Tablet  300
```

### Summary

Function	Purpose
apply() / map()	Apply custom functions to DataFrame columns
groupby()	Group data for aggregation
sum(), mean(), count(), min(), max()	Perform summary statistics
pivot_table()	Reshape data into a pivot format
sort_values(), sort_index()	Sort DataFrame by values or index

## Project 1: Sales Data Analysis

### Scenario

A retail store wants to analyze its sales data to understand trends, calculate total sales, and identify best-selling products.

### Steps to Implement

1. Import Pandas and Create a Sales DataFrame
2. Apply Discounts Using Lambda Functions
3. Group Data by Category and Compute Aggregations
4. Create a Pivot Table for Regional Sales
5. Sort Data to Identify Best-Selling Products

### Step 1: Import Pandas and Create Sales DataFrame

```
import pandas as pd
```

```
# Creating a DataFrame
data = {
    'Product': ['Laptop', 'Mouse', 'Keyboard', 'Laptop', 'Mouse', 'Keyboard'],
    'Category': ['Electronics', 'Accessories', 'Accessories', 'Electronics', 'Accessories',
    'Accessories'],
    'Region': ['East', 'West', 'East', 'West', 'East', 'West'],
    'Price': [800, 50, 100, 850, 55, 120],
    'Quantity': [5, 10, 8, 3, 15, 12]
}

df = pd.DataFrame(data)
print(df)
```

**Output:**

	Product	Category	Region	Price	Quantity
0	Laptop	Electronics	East	800	5
1	Mouse	Accessories	West	50	10
2	Keyboard	Accessories	East	100	8
3	Laptop	Electronics	West	850	3
4	Mouse	Accessories	East	55	15
5	Keyboard	Accessories	West	120	12

**Step 2: Apply Discounts Using Lambda Functions**

The store offers a **10% discount** on electronics and **5% on accessories**.

```
df['Discounted Price'] = df.apply(
    lambda row: row['Price'] * 0.9 if row['Category'] == 'Electronics' else
    row['Price'] * 0.95, axis=1
)

print(df[['Product', 'Price', 'Discounted Price']])
```

**Output:**

	Product	Price	Discounted Price
0	Laptop	800	720.0
1	Mouse	50	47.5
2	Keyboard	100	95.0
3	Laptop	850	765.0
4	Mouse	55	52.25
5	Keyboard	120	114.0

**Step 3: Group Data by Category and Compute Aggregations**

Calculate total revenue, average price, and total quantity sold per category.

```
category_summary = df.groupby('Category').agg({
    'Price': 'mean',
    'Quantity': 'sum',
    'Discounted Price': 'sum'
})

print(category_summary)
```

**Output:**

	Price	Quantity	Discounted Price
Category			
Accessories	81.25	45	308.75
Electronics	825.00	8	1485.00

**Step 4: Create a Pivot Table for Regional Sales**

Analyze total revenue per product per region.

```
pivot_table = df.pivot_table(values='Discounted Price', index='Region',
columns='Product', aggfunc='sum')
print(pivot_table)
```

**Output:**

Product	Keyboard	Laptop	Mouse
Region			
East	95.0	720.0	99.75
West	114.0	765.0	47.50

### Step 5: Sort Data to Identify Best-Selling Products

Sort products based on total revenue.

```
df['Total Revenue'] = df['Quantity'] * df['Discounted Price']
df_sorted = df.sort_values(by='Total Revenue', ascending=False)

print(df_sorted[['Product', 'Total Revenue']])
```

#### Output:

	Product	Total Revenue
4	Mouse	783.75
5	Keyboard	1368.0
3	Laptop	2295.0

**Insights:** The Laptop category generates the highest revenue.

## Project 2: Employee Salary Analysis

### Scenario

A company wants to analyze employee salaries based on departments, calculate the average salary, and sort employees by earnings.

### Steps to Implement

1. Create an Employee Salary DataFrame
2. Categorize Salaries Using a Lambda Function
3. Group Employees by Department
4. Create a Pivot Table for Salary Distribution
5. Sort Employees by Salary

**Step 1: Create an Employee Salary DataFrame**

```
import pandas as pd
```

```
# Creating Employee Salary DataFrame
data = {
    'Employee': ['John', 'Sarah', 'Mike', 'Anna', 'David', 'Emma'],
    'Department': ['IT', 'HR', 'Finance', 'IT', 'Finance', 'HR'],
    'Salary': [70000, 50000, 80000, 75000, 85000, 52000],
    'Experience': [5, 3, 8, 6, 9, 4]
}

df = pd.DataFrame(data)
print(df)
```

**Output:**

	Employee	Department	Salary	Experience
0	John	IT	70000	5
1	Sarah	HR	50000	3
2	Mike	Finance	80000	8
3	Anna	IT	75000	6
4	David	Finance	85000	9
5	Emma	HR	52000	4

**Step 2: Categorize Salaries Using a Lambda Function**

Define salary levels.

```
df['Salary Category'] = df['Salary'].apply(lambda x: 'High' if x > 75000 else
'Medium' if x > 55000 else 'Low')
print(df[['Employee', 'Salary', 'Salary Category']])
```

**Output:**

	Employee	Salary	Salary Category
0	John	70000	Medium
1	Sarah	50000	Low
2	Mike	80000	High
3	Anna	75000	Medium
4	David	85000	High
5	Emma	52000	Low

**Step 3: Group Employees by Department**

Calculate the average salary per department.

```
salary_summary = df.groupby('Department')['Salary'].mean()
print(salary_summary)
```

**Output:**

Department	Salary
Finance	82500.0
HR	51000.0
IT	72500.0

Name: Salary, dtype: float64

**Step 4: Create a Pivot Table for Salary Distribution**

```
pivot = df.pivot_table(values='Salary', index='Department', columns='Salary
Category', aggfunc='count')
print(pivot)
```

**Output:**

Salary Category	High	Low	Medium
Department			
Finance	2	0	0

HR	0	2	0
IT	0	0	2

### **Step 5: Sort Employees by Salary**

```
df_sorted = df.sort_values(by='Salary', ascending=False)
print(df_sorted[['Employee', 'Salary']])
```

#### **Output:**

	Employee	Salary
4	David	85000
2	Mike	80000
3	Anna	75000
0	John	70000
5	Emma	52000
1	Sarah	50000

#### **Insights:**

- Finance employees have the highest average salary.
- HR employees have the lowest salaries.

### **Conclusion**

These projects demonstrate data transformation and aggregation techniques like `apply()`, `groupby()`, `pivot_table()`, and `sort_values()`. These methods are essential for analyzing sales, employee data, and business trends.

## Mini Project: Customer Purchase Behavior Analysis

### Scenario

A supermarket chain wants to analyze customer purchase behavior by transforming and aggregating sales data. The goal is to understand product sales, customer spending habits, and regional trends.

### Day 51 Tasks

1. Load the dataset containing customer purchases, including Customer\_ID, Product, Category, Region, Price, Quantity, and Date\_of\_Purchase.
2. Apply a discount function using apply() and lambda:
  - a. 10% discount on Electronics
  - b. 5% discount on Groceries
3. Use map() to categorize regions into "Urban" or "Rural" based on predefined conditions.
4. Convert Date\_of\_Purchase column to datetime format using pd.to\_datetime().
5. Calculate total revenue per transaction by multiplying Price and Quantity using apply().
6. Group data by Category and calculate:
  - a. Total revenue
  - b. Average price
  - c. Total quantity sold
7. Group data by Customer\_ID and compute:
  - a. Total amount spent
  - b. Total number of purchases
8. Create a pivot table to compare revenue across Region and Category.
9. Sort customers by total spending in descending order using sort\_values().
10. Sort products alphabetically using sort\_index().
11. Identify the top-selling product based on total quantity sold.

12. Find customers who spent more than the average spending amount using boolean filtering.
13. Export the final aggregated dataset into a CSV file for further analysis.

**Objective:**

This project helps in customer segmentation, sales trend analysis, and identifying high-value products using apply(), groupby(), pivot\_table(), and sorting.

## **Mini Project 1: Employee Performance Analysis**

**Scenario:**

An HR department wants to analyze employee performance based on monthly working hours, project completion, and salary distribution. The dataset includes Employee\_ID, Department, Monthly\_Hours\_Worked, Projects\_Completed, Salary, and Joining\_Date.

**Key Objectives:**

- Use apply() to classify employees as "Underperforming," "Average," or "High Performer."
- Group employees by Department to analyze average working hours and completed projects.
- Aggregate salary data (total, average, and max salary) by department.
- Sort employees based on Projects\_Completed in descending order.
- Use pivot tables to compare average salary by department and performance classification.

## Mini Project 2: Online Store Sales Analysis

### Scenario:

An e-commerce business wants to analyze sales trends, customer spending behavior, and top-selling products. The dataset includes Order\_ID, Customer\_ID, Product\_Name, Category, Order\_Amount, Quantity, Purchase\_Date, and City.

### Key Objectives:

- Use apply() and lambda to classify orders as "Low," "Medium," or "High Value" based on Order\_Amount.
- Group orders by Category and aggregate total revenue, average order amount, and total quantity sold.
- Group customers by City to analyze spending trends in different locations.
- Create a pivot table to compare total revenue by Category and City.
- Sort the dataset to find the top-selling products based on total quantity sold.

## Day 51

### Working with Different File Formats in Pandas

#### Introduction

Pandas provides powerful tools to read and write different file formats, allowing us to work with data stored in CSV, Excel, JSON, and SQL databases. These capabilities help in data cleaning, analysis, and processing for real-world applications.

## 1. Reading & Writing CSV Files

### What is a CSV File?

- **CSV (Comma-Separated Values)** is a simple text file where data is stored in a table format, separated by commas.
- It is one of the most commonly used file formats in data analysis.

### Syntax:

```
import pandas as pd
```

```
# Reading a CSV file  
df = pd.read_csv("data.csv")
```

```
# Writing to a CSV file  
df.to_csv("output.csv", index=False)
```

### Step-by-Step Implementation

#### 1. Create a Sample CSV File

We first create a sample CSV file (`sales_data.csv`).

#### Sample `sales_data.csv` file:

```
Product,Category,Price,Quantity  
Laptop,Electronics,800,5  
Phone,Electronics,500,10  
Shoes,Fashion,50,20  
Watch,Fashion,120,8
```

## 2. Read CSV File

```
import pandas as pd

# Read CSV file
df = pd.read_csv("sales_data.csv")

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

### Output:

	Product	Category	Price	Quantity
0	Laptop	Electronics	800	5
1	Phone	Electronics	500	10
2	Shoes	Fashion	50	20
3	Watch	Fashion	120	8

## 3. Write CSV File

```
# Save dataframe to a new CSV file without index
df.to_csv("new_sales_data.csv", index=False)
```

## 2. Reading & Writing Excel Files

### What is an Excel File?

- Excel files (.xlsx or .xls) store data in multiple sheets and provide advanced formatting options.
- Pandas uses `read_excel()` and `to_excel()` for handling Excel files.

## Syntax:

```
# Reading an Excel file  
df = pd.read_excel("data.xlsx")  
  
# Writing to an Excel file  
df.to_excel("output.xlsx", index=False)
```

## Step-by-Step Implementation

### 1. Install Required Library

```
pip install openpyxl # Required for handling Excel files
```

### 2. Read Excel File

```
df = pd.read_excel("sales_data.xlsx")  
  
# Display data  
print(df.head())
```

### 3. Write Excel File

```
# Save dataframe to an Excel file  
df.to_excel("new_sales_data.xlsx", index=False, sheet_name="Sales Data")
```

### 3. Working with JSON Files

#### What is a JSON File?

- JSON (**JavaScript Object Notation**) is a lightweight format used for **storing and exchanging data**.
- It is widely used in **web APIs and configurations**.

#### Syntax:

```
# Reading a JSON file  
df = pd.read_json("data.json")
```

```
# Writing to a JSON file  
df.to_json("output.json", orient="records")
```

#### Step-by-Step Implementation

##### 1. Create a Sample JSON File

###### **Sample sales\_data.json file:**

```
[  
    {"Product": "Laptop", "Category": "Electronics", "Price": 800, "Quantity": 5},  
    {"Product": "Phone", "Category": "Electronics", "Price": 500, "Quantity": 10},  
    {"Product": "Shoes", "Category": "Fashion", "Price": 50, "Quantity": 20},  
    {"Product": "Watch", "Category": "Fashion", "Price": 120, "Quantity": 8}  
]
```

## 2. Read JSON File

```
df = pd.read_json("sales_data.json")

# Display data
print(df.head())
```

## 3. Write JSON File

```
df.to_json("new_sales_data.json", orient="records", indent=4)
```

## 4. Working with SQL Databases

### What is SQL?

- SQL (**Structured Query Language**) is used for **storing, querying, and managing relational databases**.
- Pandas can interact with SQL databases using `read_sql()` and `to_sql()`.

### Syntax:

```
import sqlite3

# Create a database connection
conn = sqlite3.connect("sales.db")

# Read data from SQL table
df = pd.read_sql("SELECT * FROM sales_data", conn)

# Write DataFrame to SQL table
df.to_sql("new_sales_data", conn, if_exists="replace", index=False)
```

## Step-by-Step Implementation

### 1. Create a Sample Database & Table

```
import sqlite3

# Connect to SQLite database (or create it)
conn = sqlite3.connect("sales.db")
# Create a cursor object
cursor = conn.cursor()

# Create a table
cursor.execute("""
CREATE TABLE IF NOT EXISTS sales_data (
    Product TEXT,
    Category TEXT,
    Price INTEGER,
    Quantity INTEGER
)
""")

# Insert sample data
cursor.execute("INSERT INTO sales_data VALUES ('Laptop', 'Electronics', 800, 5)")
cursor.execute("INSERT INTO sales_data VALUES ('Phone', 'Electronics', 500, 10)")
cursor.execute("INSERT INTO sales_data VALUES ('Shoes', 'Fashion', 50, 20)")
cursor.execute("INSERT INTO sales_data VALUES ('Watch', 'Fashion', 120, 8)")

# Commit changes and close connection
conn.commit()
conn.close()
```

## 2. Read Data from SQL

```
conn = sqlite3.connect("sales.db")

df = pd.read_sql("SELECT * FROM sales_data", conn)

print(df.head())
```

## 3. Write Data to SQL

```
df.to_sql("new_sales_data", conn, if_exists="replace", index=False)
```

## 5. Handling Large Datasets Efficiently

- When working with large files, memory management is important.
- We use:
  - chunksize → Process data in smaller chunks.
  - memory\_usage() → Check memory usage.

## Step-by-Step Implementation

### 1. Reading Large CSV with Chunksize

```
chunk_size = 1000 # Read 1000 rows at a time
chunks = pd.read_csv("large_data.csv", chunksize=chunk_size)
```

for chunk in chunks:

```
    print(chunk.head()) # Process each chunk separately
```

## 2. Check Memory Usage

```
df = pd.read_csv("large_data.csv")

print(df.memory_usage(deep=True))
```

### Real-Life Applications

Use Case	File Format	Application
Sales Data Analysis	CSV, Excel	E-commerce, Retail
API Data Processing	JSON	Web Development, Machine Learning
Database Queries	SQL	Enterprise Applications, Banking
Handling Big Data	Large CSV	Data Science, Analytics

### Summary

Function	Usage
read_csv()	Read CSV file
to_csv()	Write to CSV
read_excel()	Read Excel file
to_excel()	Write to Excel
read_json()	Read JSON file
to_json()	Write to JSON
read_sql()	Read SQL database
to_sql()	Write to SQL database
chunksize	Read large files in chunks
memory_usage()	Check memory consumption

## Conclusion

Mastering file handling in Pandas enables seamless data import, storage, and processing across different formats. Whether working with small CSV files or large SQL databases, these techniques improve efficiency and scalability in real-world projects.

## Mini Project 1: Sales Data Processing System

### Objective:

Develop a Sales Data Processing System that reads sales data from different file formats (CSV, Excel, JSON, and SQL), performs data transformations, and exports the cleaned data to a new format.

### Step 1: Install Required Libraries

```
pip install pandas openpyxl sqlite3
```

### Step 2: Read Sales Data from Different Formats

#### 1. Read from CSV File

#### Sample sales\_data.csv file:

OrderID	Product	Category	Price	Quantity
1001	Laptop	Electronics	800	2
1002	Phone	Electronics	500	5
1003	Shoes	Fashion	60	4
1004	Watch	Fashion	150	3

## Read CSV File & Display Data

```
import pandas as pd

# Read sales data from CSV
df_csv = pd.read_csv("sales_data.csv")

print(df_csv)
```

### 2. Read from Excel File

#### Sample sales\_data.xlsx file:

- **Sheet Name:** "SalesSheet"
- **Columns:** OrderID, Product, Category, Price, Quantity

## Read Excel File

```
df_excel = pd.read_excel("sales_data.xlsx", sheet_name="SalesSheet")

print(df_excel)
```

### 3. Read from JSON File

#### ❖ Sample sales\_data.json file:

```
[
    {"OrderID": 1001, "Product": "Laptop", "Category": "Electronics", "Price": 800,
     "Quantity": 2},
    {"OrderID": 1002, "Product": "Phone", "Category": "Electronics", "Price": 500,
     "Quantity": 5},
    {"OrderID": 1003, "Product": "Shoes", "Category": "Fashion", "Price": 60,
     "Quantity": 4},
```

```
{"OrderID": 1004, "Product": "Watch", "Category": "Fashion", "Price": 150,  
"Quantity": 3}  
]
```

### Read JSON File

```
df_json = pd.read_json("sales_data.json")  
  
print(df_json)
```

### 4. Read from SQL Database

#### Create & Insert Data into SQLite Database

```
import sqlite3  
  
# Connect to SQLite Database  
conn = sqlite3.connect("sales.db")  
cursor = conn.cursor()  
  
# Create Table  
cursor.execute("""  
CREATE TABLE IF NOT EXISTS sales_data (  
    OrderID INTEGER,  
    Product TEXT,  
    Category TEXT,  
    Price INTEGER,  
    Quantity INTEGER  
)  
""")
```

```
# Insert Data
cursor.execute("INSERT INTO sales_data VALUES (1001, 'Laptop', 'Electronics',
800, 2)")
cursor.execute("INSERT INTO sales_data VALUES (1002, 'Phone', 'Electronics',
500, 5)")
cursor.execute("INSERT INTO sales_data VALUES (1003, 'Shoes', 'Fashion', 60, 4)")
cursor.execute("INSERT INTO sales_data VALUES (1004, 'Watch', 'Fashion', 150,
3)")

conn.commit()
```

### Read Data from SQL

```
df_sql = pd.read_sql("SELECT * FROM sales_data", conn)

print(df_sql)
```

### Step 3: Merge and Process Data

```
# Merge all DataFrames
df = pd.concat([df_csv, df_excel, df_json, df_sql])

# Drop duplicate rows
df = df.drop_duplicates()

# Sort data by Price
df = df.sort_values(by="Price", ascending=False)

print(df)
```

## Step 4: Export Cleaned Data to Different Formats

```
# Save to CSV
```

```
df.to_csv("cleaned_sales_data.csv", index=False)
```

```
# Save to Excel
```

```
df.to_excel("cleaned_sales_data.xlsx", index=False,  
sheet_name="ProcessedSales")
```

```
# Save to JSON
```

```
df.to_json("cleaned_sales_data.json", orient="records", indent=4)
```

```
# Save to SQL
```

```
df.to_sql("cleaned_sales_data", conn, if_exists="replace", index=False)
```

### Final Output:

- cleaned\_sales\_data.csv
- cleaned\_sales\_data.xlsx
- cleaned\_sales\_data.json
- SQL table: cleaned\_sales\_data

## Mini Project 2: Large Employee Dataset Processing

### Objective:

Process large employee data stored in a CSV file efficiently using chunksize and memory\_usage(), clean the data, and store the processed data in a database.

## Step 1: Create a Large Employee CSV File (Example)

### Sample employees.csv file (1 million+ records)

```
EmplID,Name,Department,Salary,Experience
101,John,IT,60000,5
102,Susan,HR,55000,3
103,David,Finance,70000,7
104,Emma,Marketing,50000,2
```

## Step 2: Efficiently Read Large CSV File Using Chunksize

```
chunk_size = 100000 # Read 100,000 rows at a time
```

```
chunks = pd.read_csv("employees.csv", chunksize=chunk_size)
```

```
for chunk in chunks:
```

```
    print(chunk.head()) # Process each chunk separately
```

## Step 3: Check Memory Usage

```
df = pd.read_csv("employees.csv")
```

```
print(df.memory_usage(deep=True))
```

## Step 4: Clean and Process the Data

```
# Read large file in chunks and process
```

```
chunks = pd.read_csv("employees.csv", chunksize=chunk_size)
```

```
processed_chunks = []
```

```
for chunk in chunks:
```

```
    # Remove duplicate records
```

```
    chunk = chunk.drop_duplicates()
```

```
# Convert Salary to Integer
chunk["Salary"] = chunk["Salary"].astype(int)

# Standardize Department Names (capitalize)
chunk["Department"] = chunk["Department"].str.capitalize()

processed_chunks.append(chunk)

# Combine processed data
df_processed = pd.concat(processed_chunks)

print(df_processed.head())
```

### **Step 5: Store Cleaned Data in SQL Database**

```
conn = sqlite3.connect("employees.db")

df_processed.to_sql("processed_employees", conn, if_exists="replace",
index=False)
```

#### **Final Output:**

- Processed Data in processed\_employees SQL Table
- Optimized data storage using chunksize and memory\_usage()

### **Real-Life Mini Project: Employee Data Management System**

#### **Objective:**

Create an Employee Data Management System that handles employee records stored in various file formats (CSV, Excel, JSON, SQL). The project will involve data

processing, transformation, and exporting to different formats while efficiently handling large datasets.

### Day 51 Tasks :

1. Read employee data from a CSV file using `read_csv()` and display the first five rows.
2. Read employee data from an Excel file using `read_excel()` and extract specific columns (Employee ID, Name, Department).
3. Read employee data from a JSON file using `read_json()` and display basic statistics about the dataset.
4. Store employee records in an SQLite database and fetch all records using `read_sql()`.
5. Merge data from multiple sources (CSV, Excel, JSON, SQL) into a single DataFrame.
6. Remove duplicate records using `drop_duplicates()` and count the number of unique employees.
7. Fill missing values in the salary column using `fillna()` with the department's average salary.
8. Convert salary data type to integer using `astype()`.
9. Change column names to standard format using `rename()`, ensuring consistent naming conventions.
10. Sort employees based on experience and salary using `sort_values()`.
11. Export the cleaned dataset to a new CSV file using `to_csv()`.
12. Export the dataset to an Excel file with multiple sheets using `to_excel()`.
13. Process large datasets efficiently by reading the CSV file in chunks (`chunksize`) and checking memory usage (`memory_usage()`).

## Mini Projects :

### 1. Sales Data Processing System

**Objective:** Develop a system that processes sales data from different file formats (CSV, Excel, JSON, SQL). The system will handle data extraction, transformation, and efficient storage for business analytics.

#### Key Features:

- Read sales data from CSV, Excel, and JSON.
- Store and retrieve sales records from an SQL database.
- Identify missing values and apply proper handling techniques.
- Sort and filter sales data based on date, product category, and revenue.
- Export cleaned and processed data into multiple formats (CSV, Excel, JSON).
- Efficiently handle large datasets using chunksize and memory optimization techniques.

### 2. Customer Feedback Analysis System

**Objective:** Build a system that processes customer feedback and reviews stored in various file formats for sentiment analysis and business insights.

#### Key Features:

- Read customer feedback from CSV, Excel, and JSON.
- Convert textual feedback into structured DataFrame format.
- Handle missing values in customer responses.
- Remove duplicate feedback using drop\_duplicates().
- Store structured feedback in an SQL database for further analysis.
- Export cleaned customer feedback to CSV, Excel, and JSON for reporting.
- Process large datasets efficiently using chunksize and memory\_usage().

# Day 52

## Merging, Joining & Concatenating Data in Pandas

When working with real-world data, you often need to combine multiple datasets. Pandas provides powerful methods to **merge, join, and concatenate** DataFrames efficiently.

### Concatenating DataFrames (`concat()`)

#### Definition

Concatenation is used to stack multiple DataFrames either vertically (rows-wise) or horizontally (column-wise).

#### Syntax:

```
pd.concat([df1, df2], axis=0) # Vertical (default)  
pd.concat([df1, df2], axis=1) # Horizontal
```

### Real-Life Example: Combining Monthly Sales Data

Step 1: Import Pandas and Create Two DataFrames

```
import pandas as pd
```

```
# Sales data for January  
df_jan = pd.DataFrame({  
    'Product': ['Laptop', 'Mouse', 'Keyboard'],  
    'Sales': [100, 200, 150]  
})
```

```
# Sales data for February
df_feb = pd.DataFrame({
    'Product': ['Laptop', 'Mouse', 'Keyboard'],
    'Sales': [120, 220, 180]
})

print("January Sales Data:\n", df_jan)
print("February Sales Data:\n", df_feb)
```

Step 2: Concatenate DataFrames (Row-wise)

```
df_total = pd.concat([df_jan, df_feb], ignore_index=True)
print("Total Sales Data:\n", df_total)
```

### Output:

	Product	Sales
0	Laptop	100
1	Mouse	200
2	Keyboard	150
3	Laptop	120
4	Mouse	220
5	Keyboard	180

The data is combined **row-wise**, stacking February sales below January sales.

## Merging DataFrames (merge())

### Definition

Merging is used to **combine datasets based on a common column (key)**, similar to SQL joins.

### Syntax:

```
pd.merge(df1, df2, on='common_column', how='inner')
```

◆ how parameter:

- **inner** → Only matching rows
- **left** → All rows from left DataFrame
- **right** → All rows from right DataFrame
- **outer** → All rows from both

### Real-Life Example: Merging Customer & Purchase Data

Step 1: Create Two DataFrames

```
df_customers = pd.DataFrame({
    'CustomerID': [1, 2, 3],
    'Name': ['Alice', 'Bob', 'Charlie']
})
```

```
df_purchases = pd.DataFrame({
    'CustomerID': [1, 2, 4],
    'Product': ['Laptop', 'Mouse', 'Keyboard']
})
```

```
print("Customers Data:\n", df_customers)
print("Purchases Data:\n", df_purchases)
```

Step 2: Merge DataFrames on CustomerID

```
df_merged = pd.merge(df_customers, df_purchases, on='CustomerID',
how='inner')
print("Merged Data:\n", df_merged)
```

### **Output:**

	CustomerID	Name	Product
0	1	Alice	Laptop
1	2	Bob	Mouse

Only customers who made purchases are included (inner join).

## **Joining DataFrames (join())**

### **Definition**

Joins work like merge(), but are used when DataFrames **share an index**.

### **Syntax:**

```
df1.join(df2, how='inner')
```

**how parameter:**

- **inner** → Only matching index values
- **left** → All rows from left DataFrame

**Real-Life Example: Joining Employee Details & Salaries****Step 1: Create Two DataFrames with Indexes**

```
df_employees = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Department': ['HR', 'IT', 'Finance']
}, index=[101, 102, 103])
```

```
df_salaries = pd.DataFrame({
    'Salary': [50000, 60000, 55000]
}, index=[101, 102, 103])
```

```
print("Employee Data:\n", df_employees)
print("Salary Data:\n", df_salaries)
```

**Step 2: Join DataFrames on Index**

```
df_joined = df_employees.join(df_salaries)
print("Joined Data:\n", df_joined)
```

**Output:**

	Name	Department	Salary
101	Alice	HR	50000
102	Bob	IT	60000
103	Charlie	Finance	55000

DataFrames are merged based on their index values.

## Handling Duplicate Columns After Merging

### Definition

When merging, sometimes columns with the same name appear twice. We can resolve this using suffixes.

### Syntax:

```
pd.merge(df1, df2, on='common_column', suffixes=('_left', '_right'))
```

### Real-Life Example: Avoiding Duplicate Columns

Step 1: Create DataFrames with Overlapping Column Names

```
df_a = pd.DataFrame({  
    'ID': [1, 2],  
    'Name': ['Alice', 'Bob'],  
    'Age': [25, 30]  
})  
  
df_b = pd.DataFrame({  
    'ID': [1, 2],  
    'Age': [26, 31], # Different Age column  
    'Salary': [50000, 60000]  
})  
  
print("DataFrame A:\n", df_a)  
print("DataFrame B:\n", df_b)
```

## Step 2: Merge and Handle Duplicate Columns

```
df_clean = pd.merge(df_a, df_b, on='ID', suffixes=('_A', '_B'))
print("Merged Data:\n", df_clean)
```

### Output:

	ID	Name	Age_A	Age_B	Salary
0	1	Alice	25	26	50000
1	2	Bob	30	31	60000

✓ Now, we can clearly differentiate between the **Age columns from both DataFrames**.

### Summary

Method	Purpose
concat()	Stack DataFrames (rows or columns)
merge()	Combine DataFrames based on a key (like SQL JOIN)
join()	Merge DataFrames using the index
suffixes=('_A', '_B')	Avoid duplicate column names

## Mini Project 1: Customer Orders & Product Details Integration

### Problem Statement

A retail company has two datasets:

1. Customer Orders (contains customer ID and product ID)
2. Product Details (contains product ID and product name)

Your task is to merge these datasets and concatenate new monthly order records.

### Step 1: Import Necessary Libraries

```
import pandas as pd
```

### Step 2: Create DataFrames for Customer Orders & Product Details

```
# Customer Orders DataFrame
df_orders = pd.DataFrame({
    'OrderID': [101, 102, 103, 104],
    'CustomerID': [1, 2, 3, 4],
    'ProductID': [201, 202, 203, 204]
})
```

```
# Product Details DataFrame
df_products = pd.DataFrame({
    'ProductID': [201, 202, 203, 204, 205],
    'ProductName': ['Laptop', 'Mouse', 'Keyboard', 'Monitor', 'Headset']
})
```

```
print("Customer Orders:\n", df_orders)
print("Product Details:\n", df_products)
```

### Step 3: Merge Orders with Product Details (Using merge())

```
df_merged = pd.merge(df_orders, df_products, on='ProductID', how='inner')
print("Merged Orders with Product Details:\n", df_merged)
```

## Explanation

- The merge() function is used to combine df\_orders and df\_products based on ProductID.
- We use how='inner' to keep only matching product IDs.

## Output

	OrderID	CustomerID	ProductID	ProductName
0	101	1	201	Laptop
1	102	2	202	Mouse
2	103	3	203	Keyboard
3	104	4	204	Monitor

Step 4: Concatenate New Monthly Orders (Using concat())

```
# New Orders for the next month
df_new_orders = pd.DataFrame({
    'OrderID': [105, 106],
    'CustomerID': [5, 6],
    'ProductID': [205, 202]
})

# Concatenate both order datasets
df_all_orders = pd.concat([df_orders, df_new_orders], ignore_index=True)
print("Updated Orders:\n", df_all_orders)
```

## Explanation

- New orders for the next month are concatenated with previous orders.
- ignore\_index=True resets the index.

## Output

	OrderID	CustomerID	ProductID
0	101	1	201
1	102	2	202
2	103	3	203
3	104	4	204
4	105	5	205
5	106	6	202

## Step 5: Handle Duplicate Columns After Merging

```
# Simulate another Product DataFrame with duplicate column names
df_product_prices = pd.DataFrame({
    'ProductID': [201, 202, 203, 204, 205],
    'ProductName': ['Laptop', 'Mouse', 'Keyboard', 'Monitor', 'Headset'],
    'Price': [800, 20, 50, 150, 30]
})

# Merge with suffixes to handle duplicate column names
df_final = pd.merge(df_merged, df_product_prices, on='ProductID',
                     suffixes=('_Order', '_Details'))
print("Final Merged Data:\n", df_final)
```

## Explanation

- The ProductName column appears in both DataFrames, so we use suffixes=('\_Order', '\_Details').

## Output

	OrderID	CustomerID	ProductID	ProductName_Order	ProductName_Details	Price
0	101	1	201	Laptop	Laptop	800
1	102	2	202	Mouse	Mouse	20
2	103	3	203	Keyboard	Keyboard	50
3	104	4	204	Monitor	Monitor	150

Project Completed! We successfully merged customer orders with product details and concatenated new orders.

## Mini Project 2: Employee Database Integration

### Problem Statement

A company maintains two datasets:

1. Employee Details (contains employee ID, name, department)
2. Employee Salary (contains employee ID and salary)

Your task is to join these datasets using the employee ID and merge new employee records.

### Step 1: Create DataFrames for Employee Details & Salary

```
df_employees = pd.DataFrame({
    'EmpID': [101, 102, 103, 104],
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Department': ['HR', 'IT', 'Finance', 'Marketing']
})
```

```
df_salary = pd.DataFrame({  
    'EmpID': [101, 102, 103, 104],  
    'Salary': [50000, 60000, 55000, 48000]  
})  
  
print("Employee Details:\n", df_employees)  
print("Employee Salary:\n", df_salary)
```

### Step 2: Join Employee Details with Salary (Using join())

```
df_employees.set_index('EmpID', inplace=True)  
df_salary.set_index('EmpID', inplace=True)  
  
df_final = df_employees.join(df_salary)  
print("Joined Employee Data:\n", df_final)
```

### Explanation

- Both DataFrames use EmpID as an index before joining.
- The join() function merges data based on the index.

### Output

	Name	Department	Salary
EmplID			
101	Alice	HR	50000
102	Bob	IT	60000
103	Charlie	Finance	55000
104	David	Marketing	48000

### Step 3: Concatenate New Employee Records

```
df_new_employees = pd.DataFrame({
    'EmpID': [105, 106],
    'Name': ['Eve', 'Frank'],
    'Department': ['IT', 'HR']
}).set_index('EmpID')
```

```
df_all_employees = pd.concat([df_final, df_new_employees], ignore_index=False)
print("Updated Employee Data:\n", df_all_employees)
```

### Output

	Name	Department	Salary
EmpID			
101	Alice	HR	50000
102	Bob	IT	60000
103	Charlie	Finance	55000
104	David	Marketing	48000
105	Eve	IT	NaN
106	Frank	HR	NaN

New employees are **concatenated** into the table.

### Step 4: Handle Missing Salaries (Using fillna())

```
df_all_employees['Salary'] = df_all_employees['Salary'].fillna(40000)
print("Final Employee Data (Missing Salaries Filled):\n", df_all_employees)
```

## Output

	Name	Department	Salary
EmplID			
101	Alice	HR	50000
102	Bob	IT	60000
103	Charlie	Finance	55000
104	David	Marketing	48000
105	Eve	IT	40000
106	Frank	HR	40000

Missing salaries were filled with a default value of 40000.

## Summary

Concept	Used In
merge()	Merging Customer Orders with Product Details
concat()	Adding New Orders & Employees
join()	Joining Employee Details with Salaries
fillna()	Handling Missing Values in Salary

## Real-Life Mini Project: University Student & Course Data Integration

### Problem Statement:

A university manages multiple datasets containing student information, enrolled courses, and faculty details. The goal is to merge, join, and concatenate these datasets to create a unified student database.

## Day 52 Tasks

1. Create a Student DataFrame containing StudentID, Name, Department.
2. Create a Course Enrollment DataFrame with StudentID, CourseID, CourseName.
3. Create a Faculty DataFrame with CourseID, FacultyName.
4. Merge Student Data with Course Enrollment using merge().
5. Merge Course Enrollment with Faculty Details using merge().
6. Concatenate New Student Enrollment Records using concat().
7. Join Student Data with Course Enrollment using join().
8. Handle duplicate column names after merging using suffixes=().
9. Sort the merged dataset based on StudentID using sort\_values().
10. Filter out students from a specific department (e.g., Computer Science).
11. Group data by Department and count students using groupby().
12. Fill missing values (NaN) for students without course enrollments using fillna().
13. Export the final cleaned DataFrame to a CSV file (to\_csv()).

This real-world university database integration project will help understand merging, joining, and concatenating in data science and data engineering.

## Real-Life Mini Project Requirements

### 1. Sales & Customer Data Integration System

- A retail company wants to merge and analyze its sales and customer data.
- Data Sources:
  - Customer Details (CustomerID, Name, Email, Location)
  - Purchase History (CustomerID, OrderID, Product, Amount)
  - Product Details (ProductID, Product, Category, Price)
- Key Operations:
  - Merge sales and customer data to see who purchased what.

- Join product details with the purchase history.
- Concatenate new customer records into the dataset.
- Handle duplicate columns after merging multiple datasets.
- Export the final integrated dataset for business analysis.

## 2. Hospital Patient & Doctor Data Integration

- A hospital wants to combine patient, doctor, and appointment records to create a centralized database.
- Data Sources:
  - Patient Details (PatientID, Name, Age, Diagnosis)
  - Doctor Information (DoctorID, DoctorName, Specialization)
  - Appointment Records (PatientID, DoctorID, Date, Prescription)
- Key Operations:
  - Merge patient and appointment records to track medical history.
  - Join doctor details with appointment records.
  - Concatenate new patient registrations into the dataset.
  - Handle duplicate columns while merging multiple tables.
  - Generate a final structured report for medical analysis.

# Day 53

## Advanced Pandas & Performance Optimization

### 1. MultiIndexing & Hierarchical Indexing

#### Definition

MultiIndexing allows handling multi-level row and column labels in Pandas DataFrames, making it easier to organize hierarchical data (e.g., sales data by region and year).

## Syntax

```
df.set_index(['Column1', 'Column2'])
df.reset_index()
```

### Real-Life Example: Sales Data Analysis

**Problem Statement:** A company wants to analyze sales data grouped by Region and Year.

#### Step 1: Create a MultiIndex DataFrame

```
import pandas as pd
```

```
# Sample Data
data = {
    'Region': ['North', 'North', 'South', 'South'],
    'Year': [2022, 2023, 2022, 2023],
    'Sales': [10000, 15000, 12000, 18000]
}
```

```
df = pd.DataFrame(data)
```

```
# Setting MultiIndex
df.set_index(['Region', 'Year'], inplace=True)
print(df)
```

## 2. Reshaping Data (**melt()**, **stack()**, **unstack()**)

### Definition

- **melt():** Converts wide format to long format.
- **stack():** Converts columns into a hierarchical row index.

- `unstack()`: Converts row index levels into columns.

### Syntax

```
df.melt(id_vars=['col1'], value_vars=['col2', 'col3'])
df.stack()
df.unstack()
```

### Real-Life Example: Reshaping Monthly Sales Data

**Problem Statement:** Convert sales data where months are columns into a long format.

#### Step 1: Create & Melt Data

```
import pandas as pd
```

```
# Sample Data
df = pd.DataFrame({
    'Product': ['A', 'B'],
    'Jan': [200, 150],
    'Feb': [220, 180]
})

# Convert columns to rows using melt
df_melted = df.melt(id_vars=['Product'], var_name='Month', value_name='Sales')
print(df_melted)
```

## 3. Window Functions (`rolling()`, `expanding()`)

### Definition

- `rolling()`: Applies calculations over a sliding window (e.g., moving average).
- `expanding()`: Expands the dataset and applies cumulative calculations.

## Syntax

```
df['column'].rolling(window=3).mean()
df['column'].expanding().sum()
```

Real-Life Example: Moving Average of Stock Prices

**Problem Statement:** Calculate a 3-day moving average for stock prices.

### Step 1: Create & Apply Rolling Window

```
import pandas as pd
```

```
# Sample Data
df = pd.DataFrame({'Day': [1, 2, 3, 4, 5], 'Stock Price': [100, 102, 105, 107, 110]})

# Compute rolling mean
df['Moving_Avg'] = df['Stock Price'].rolling(window=3).mean()
print(df)
```

## 4. Performance Optimization

### Definition

- Vectorization vs Loops: Using NumPy & Pandas built-in functions is faster than Python loops.
- .iterrows() vs .apply(): apply() is faster for row-wise operations than iterrows().

## Syntax

```
df['new_col'] = df['col'].apply(lambda x: x * 2)
```

Real-Life Example: Optimizing Salary Increment Calculation

**Problem Statement:** Apply a 10% salary hike using vectorization instead of loops.

### Step 1: Using apply() Instead of Loops

```
import pandas as pd
```

```
# Sample Data
```

```
df = pd.DataFrame({'Employee': ['A', 'B', 'C'], 'Salary': [50000, 60000, 70000]})
```

```
# Vectorized operation
```

```
df['New Salary'] = df['Salary'].apply(lambda x: x * 1.10)
```

```
print(df)
```

## 5. Pandas with Matplotlib (Data Visualization)

### Definition

Pandas integrates with Matplotlib to generate plots like line charts, bar charts, and histograms.

### Syntax

```
df.plot(kind='bar')
```

Real-Life Example: Visualizing Monthly Sales

**Problem Statement:** Plot a bar chart of monthly sales.

### Step 1: Create & Plot Data

```
import pandas as pd
```

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

```
# Sample Data
```

```
df = pd.DataFrame({'Month': ['Jan', 'Feb', 'Mar'], 'Sales': [200, 250, 300]})
```

```
# Plot
df.plot(x='Month', y='Sales', kind='bar', title="Monthly Sales", color='blue')
plt.show()
```

## Summary

Concept	Key Function	Real-Life Example
MultiIndexing	set_index(), reset_index()	Sales by Region & Year
Reshaping Data	melt(), stack(), unstack()	Transform Monthly Sales Data
Window Functions	rolling(), expanding()	Stock Price Moving Average
Performance Optimization	apply(), vectorization	Salary Increment Calculation
Data Visualization	plot()	Monthly Sales Chart

## Real-Life Mini Project 1: Employee Performance Analysis

### Project Description

A company wants to analyze employee productivity using hierarchical indexing, data reshaping, window functions, performance optimization, and visualization.

The dataset contains:

- Department
- Employee Name
- Monthly Productivity Score
- Years of Experience

### Step 1: Import Necessary Libraries

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

**Step 2: Create a Sample Dataset**

```
data = {
    'Department': ['HR', 'HR', 'IT', 'IT', 'Finance', 'Finance'],
    'Employee': ['Alice', 'Bob', 'Charlie', 'David', 'Eve', 'Frank'],
    'Jan Productivity': [78, 85, 90, 88, 76, 80],
    'Feb Productivity': [80, 87, 92, 89, 78, 83],
    'Mar Productivity': [85, 88, 94, 90, 79, 85],
    'Years of Experience': [3, 5, 8, 6, 4, 7]
}

df = pd.DataFrame(data)
print(df)
```

**Step 3: Apply MultiIndexing (Hierarchical Indexing)**

```
df.set_index(['Department', 'Employee'], inplace=True)
print(df)
```

This groups employees by their department.

**Step 4: Reshaping Data Using melt()**

```
df_reset = df.reset_index()
df_melted = df_reset.melt(id_vars=['Department', 'Employee', 'Years of
Experience'],
                           var_name='Month', value_name='Productivity')
print(df_melted)
```

This converts the wide format (separate columns for months) into a long format.

**Step 5: Using Window Functions (rolling())**

```
df_melted['Rolling Avg'] =
df_melted.groupby('Employee')['Productivity'].rolling(window=2).mean().reset_index(0, drop=True)
print(df_melted)
```

This calculates a 2-month rolling average productivity for each employee.

**Step 6: Performance Optimization****Vectorization vs Loops**

```
# Using a Loop (Slower)
df_melted['Experience Bonus'] = 0
for i in range(len(df_melted)):
    df_melted.loc[i, 'Experience Bonus'] = df_melted.loc[i, 'Years of Experience'] *
100
```

```
# Using Apply (Optimized)
df_melted['Experience Bonus'] = df_melted['Years of Experience'].apply(lambda x:
x * 100)
print(df_melted)
```

Using apply() instead of a loop speeds up calculations.

**Step 7: Data Visualization with Matplotlib**

```
plt.figure(figsize=(10,5))
for emp in df_melted['Employee'].unique():
    subset = df_melted[df_melted['Employee'] == emp]
    plt.plot(subset['Month'], subset['Productivity'], marker='o', label=emp)

plt.title("Employee Productivity Over 3 Months")
plt.xlabel("Month")
```

```
plt.ylabel("Productivity Score")
plt.legend()
plt.show()
```

This visualizes employee productivity trends.

## Summary of Mini Project 1

Concept	Function Used
MultiIndexing	set_index()
Reshaping Data	melt()
Window Functions	rolling()
Performance Optimization	apply() instead of loop
Visualization	Matplotlib plot()

## Real-Life Mini Project 2: Sales Performance Analysis

### Project Description

A retail company wants to analyze sales trends per region and product category using multi-indexing, reshaping, window functions, and performance optimization.

### Step 1: Import Libraries

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

**Step 2: Create a Sample Dataset**

```

sales_data = {
    'Region': ['North', 'North', 'South', 'South', 'West', 'West'],
    'Category': ['Electronics', 'Furniture', 'Electronics', 'Furniture', 'Electronics',
    'Furniture'],
    'Jan Sales': [10000, 12000, 15000, 17000, 11000, 13000],
    'Feb Sales': [11000, 13000, 16000, 18000, 12000, 14000],
    'Mar Sales': [12000, 13500, 17000, 18500, 13000, 14500]
}

```

```

df_sales = pd.DataFrame(sales_data)
print(df_sales)

```

**Step 3: Apply MultiIndexing**

```

df_sales.set_index(['Region', 'Category'], inplace=True)
print(df_sales)

```

This groups sales data by Region and Category.

**Step 4: Reshaping Data Using melt()**

```

df_sales_reset = df_sales.reset_index()
df_sales_melted = df_sales_reset.melt(id_vars=['Region', 'Category'],
                                       var_name='Month', value_name='Sales')
print(df_sales_melted)

```

This transforms the dataset from wide to long format.

**Step 5: Using Window Functions (rolling())**

```
df_sales_melted['3-Month Avg Sales'] =
df_sales_melted.groupby('Category')['Sales'].rolling(window=3).mean().reset_index(0, drop=True)
print(df_sales_melted)
```

This calculates a 3-month rolling average for sales.

**Step 6: Performance Optimization****Using Apply Instead of Loops**

```
df_sales_melted['Discounted Price'] = df_sales_melted['Sales'].apply(lambda x: x
* 0.95)
print(df_sales_melted)
```

Instead of using loops, apply() is used for faster computation.

**Step 7: Data Visualization with Matplotlib**

```
plt.figure(figsize=(10,5))
for category in df_sales_melted['Category'].unique():
    subset = df_sales_melted[df_sales_melted['Category'] == category]
    plt.plot(subset['Month'], subset['Sales'], marker='o', label=category)

plt.title("Sales Performance Over 3 Months")
plt.xlabel("Month")
plt.ylabel("Sales Revenue")
plt.legend()
plt.show()
```

This visualizes sales performance trends for different categories.

## Summary of Mini Project 2

Concept	Function Used
MultiIndexing	set_index()
Reshaping Data	melt()
Window Functions	rolling()
Performance Optimization	apply() instead of loop
Visualization	Matplotlib plot()

## Final Takeaways

**Both projects demonstrate:**

- MultiIndexing for hierarchical data.
- Reshaping (melt) to transform data.
- Rolling windows for trend analysis.
- Performance optimization using apply().
- Matplotlib for clear data visualization.

## Real-Life Mini Project: Stock Market Data Analysis

**Project Description:**

A financial analyst wants to analyze stock market trends using advanced Pandas & performance optimization techniques. The dataset contains daily stock prices for multiple companies.

## Day 53 Tasks:

1. Load the Stock Market Dataset

- Read a CSV file containing stock market data (columns: Date, Company, Open Price, Close Price, Volume).

## 2. Apply MultiIndexing for Companies & Dates

- Set a hierarchical index with Company and Date for efficient data analysis.

## 3. Reshape Data Using melt()

- Convert stock prices from wide format (separate Open and Close columns) to a long format.

## 4. Use stack() and unstack() to Modify Indexing

- Reshape data to swap levels in hierarchical indexing.

## 5. Apply Rolling Window Functions (rolling())

- Calculate a 7-day moving average of stock closing prices for each company.

## 6. Use Expanding Window Functions (expanding())

- Compute the cumulative maximum stock price over time.

## 7. Compare Vectorization vs Loops for Performance

- Calculate daily percentage change in stock price using both a loop and a vectorized approach.

## 8. Optimize Data Processing with apply() instead of .iterrows()

- Compute adjusted closing prices by applying a function across rows.

## 9. Compute Aggregations with groupby()

- Find the total trading volume per company for each month.

## 10. Handle Missing Data Efficiently

- Identify and fill missing stock prices using appropriate methods (fillna() or interpolation).

**11. Sort Data by Date and Company**

- Use `sort_values()` and `sort_index()` to arrange data properly.

**12. Generate Visualizations with Pandas & Matplotlib**

- Plot stock price trends for selected companies using Matplotlib.

**13. Save the Processed Data to a New File**

- Write the cleaned and transformed data to a new CSV file for future analysis.

**Real-Life Mini Project Requirements (Without Solutions)****1. Sales Performance Analysis for a Retail Chain**

A retail company wants to analyze its sales performance across multiple store locations over time. The dataset contains sales transactions with details such as Date, Store, Product Category, Sales Amount, and Quantity Sold.

**Key Requirements:**

- Use MultiIndexing to structure data by Store and Date.
- Reshape sales data using `melt()`, `stack()`, and `unstack()`.
- Apply rolling averages to compute a 7-day moving average of total sales per store.
- Optimize performance by replacing loops with vectorized operations and using `.apply()` instead of `.iterrows()`.
- Group data by Product Category to find total revenue and average sales.
- Create visualizations using Pandas & Matplotlib to identify sales trends.

## 2. Website Traffic & User Engagement Analysis

A digital marketing team wants to analyze website traffic trends and user engagement from different locations. The dataset contains user activity details such as Date, User ID, Country, Page Views, Time Spent (seconds), and Bounce Rate (%).

### Key Requirements:

- Set MultiIndexing with Country and Date for structured analysis.
- Use melt() and unstack() to reshape engagement metrics.
- Apply rolling window functions (rolling(), expanding()) to track cumulative page views and weekly average session duration.
- Compare loop-based vs vectorized calculations for bounce rate trends.
- Optimize data processing by using .apply() instead of .iterrows().
- Generate interactive line plots using Matplotlib to visualize engagement trends over time.

# Matplotlib

## Day 54

### Introduction to Matplotlib & Basic Plots

Matplotlib is one of the most widely used data visualization libraries in Python. It helps in creating static, animated, and interactive visualizations for different types of data.

## What is Matplotlib? Why use it for Data Visualization?

### Definition:

Matplotlib is a 2D plotting library that allows users to generate a wide range of graphs and charts, such as line graphs, bar charts, scatter plots, histograms, etc.

### Why Use Matplotlib?

- ✓ Helps in understanding trends and patterns in data.
- ✓ Supports multiple types of plots (line, bar, scatter, histogram, etc.).
- ✓ Can customize charts with titles, labels, legends, colors, and markers.
- ✓ Works seamlessly with NumPy, Pandas, and other libraries.
- ✓ Can export figures in multiple formats (PNG, PDF, SVG, etc.).

### Installing Matplotlib

Before using Matplotlib, we need to install it.

### Command to Install

```
pip install matplotlib
```

This will install the Matplotlib library.

### Importing Matplotlib

After installation, import matplotlib.pyplot to start plotting.

### Syntax

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

- pyplot is a module inside Matplotlib that provides simple functions for creating plots.

### Basic Plot (plot())

The plot() function is used to create a line plot.

#### Syntax

```
plt.plot(x_values, y_values, format)
```

- x\_values: X-axis data
- y\_values: Y-axis data
- format: Line color, marker style, and line style (optional)

### Real-Life Example – Plotting Monthly Sales Trend

Problem Statement:

A shop owner wants to visualize sales trends over six months.

#### Step-by-Step Solution

```
import matplotlib.pyplot as plt

# Data
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
sales = [200, 450, 300, 500, 700, 650] # Sales in dollars

# Plot the sales trend
plt.plot(months, sales, marker='o', linestyle='-', color='b')

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

## Output

A line graph showing sales increasing over six months.

## Adding Titles, Labels, Legends

To improve readability, we can add:

- Title using plt.title()
- X-axis label using plt.xlabel()
- Y-axis label using plt.ylabel()
- Legend using plt.legend()

## Real-Life Example – Comparing Sales of Two Products

Problem Statement:

A shop sells Product A and Product B. The owner wants to compare their sales trends.

Step-by-Step Solution

```
import matplotlib.pyplot as plt

# Data
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
sales_A = [200, 450, 300, 500, 700, 650] # Product A Sales
sales_B = [150, 400, 350, 450, 600, 620] # Product B Sales

# Plot both products
plt.plot(months, sales_A, marker='o', linestyle='-', color='b', label="Product A")
plt.plot(months, sales_B, marker='s', linestyle='--', color='r', label="Product B")

# Add labels, title, and legend
plt.xlabel("Months")
```

```
plt.ylabel("Sales ($)")  
plt.title("Sales Comparison of Product A and B")  
plt.legend()  
  
# Show the plot  
plt.show()
```

## Output

A graph with two different line styles representing the sales of Product A and B, along with labels and a legend.

## Displaying a Plot (show())

- plt.show() is used to display the figure on the screen.
- It must be called once at the end of plotting.

## Example

```
plt.plot(x, y)  
plt.show()
```

This command renders the graph.

## Saving a Figure (savefig())

To save the plot as an image, use:

## Syntax

```
plt.savefig("filename.png")
```

- Supports multiple formats like PNG, JPG, PDF, SVG.

### Real-Life Example

```
plt.plot(months, sales, marker='o', linestyle='-', color='b')
plt.xlabel("Months")
plt.ylabel("Sales ($)")
plt.title("Monthly Sales Trend")

# Save the plot as an image
plt.savefig("monthly_sales.png")

plt.show()
```

The image monthly\_sales.png is saved in the working directory.

### Summary of Key Functions

Function	Purpose
plt.plot()	Creates a line plot
plt.xlabel()	Adds X-axis label
plt.ylabel()	Adds Y-axis label
plt.title()	Adds title to the plot
plt.legend()	Adds a legend to differentiate data
plt.show()	Displays the plot
plt.savefig()	Saves the plot as an image

### Conclusion

Matplotlib is a powerful and easy-to-use library for visualizing data in Python.

By learning basic plots, labels, legends, and saving figures, you can start creating professional charts for your data analysis projects.

## Mini Project 1: Analyzing Website Traffic Trends

### Objective

A company wants to analyze its website traffic over six months to understand user engagement and make business decisions. The goal is to visualize the trend using Matplotlib.

### Steps to Implement

- ✓ Install & Import Matplotlib
- ✓ Define the traffic data for six months
- ✓ Create a line plot to visualize traffic
- ✓ Add title, labels, and legend
- ✓ Save the figure and display the plot

### Step 1: Install Matplotlib

Before using Matplotlib, install it using:

```
pip install matplotlib
```

### Step 2: Import Required Libraries

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

### Step 3: Define Website Traffic Data

- X-axis → Months
- Y-axis → Number of visitors per month

```
# Data
```

```
months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun"]
```

```
visitors = [5000, 7000, 8000, 12000, 15000, 18000] # Monthly website visitors
```

#### Step 4: Create a Line Plot

```
plt.plot(months, visitors, marker="o", linestyle="-", color="b", label="Website  
Visitors")
```

- `marker="o"` → Adds circle markers at each data point
- `linestyle="-"` → Creates a solid line
- `color="b"` → Sets the line color to blue
- `label="Website Visitors"` → Assigns a legend label

#### Step 5: Add Titles, Labels & Legends

```
plt.title("Monthly Website Traffic Analysis")  
plt.xlabel("Months")  
plt.ylabel("Number of Visitors")  
plt.legend()
```

- `plt.title()` → Adds a title
- `plt.xlabel()` → Labels the X-axis
- `plt.ylabel()` → Labels the Y-axis
- `plt.legend()` → Displays the legend

#### Step 6: Display & Save the Plot

```
plt.savefig("website_traffic.png") # Save the figure  
plt.show() # Display the plot
```

- `plt.savefig("website_traffic.png")` → Saves the figure as PNG
- `plt.show()` → Displays the line plot

**Full Code**

```
import matplotlib.pyplot as plt

# Data
months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun"]
visitors = [5000, 7000, 8000, 12000, 15000, 18000] # Monthly website visitors

# Create the plot
plt.plot(months, visitors, marker="o", linestyle="-", color="b", label="Website Visitors")

# Add labels, title, and legend
plt.title("Monthly Website Traffic Analysis")
plt.xlabel("Months")
plt.ylabel("Number of Visitors")
plt.legend()

# Save and show the plot
plt.savefig("website_traffic.png")
plt.show()
```

**Expected Output**

A line graph showing the trend of website visitors over six months, with titles, labels, and markers.

**Mini Project 2: Comparing Product Sales****Objective**

A business wants to compare monthly sales of two different products (Product A & Product B) and visualize their performance using Matplotlib.

## Steps to Implement

- ✓ Install & Import Matplotlib
- ✓ Define the sales data
- ✓ Plot two sales trends on the same graph
- ✓ Add title, labels, and legend
- ✓ Save the figure and display the plot

### Step 1: Install & Import Matplotlib

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

### Step 2: Define Sales Data

```
# Data
months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun"]
sales_A = [3000, 4500, 5200, 6100, 7200, 8500] # Sales of Product A
sales_B = [2800, 4000, 4700, 5900, 7100, 8000] # Sales of Product B
```

### Step 3: Create Line Plots for Both Products

```
plt.plot(months, sales_A, marker="o", linestyle="-", color="g", label="Product A Sales")
plt.plot(months, sales_B, marker="s", linestyle="--", color="r", label="Product B Sales")
```

- marker="o" → Circles for Product A
- marker="s" → Squares for Product B
- linestyle="-" → Solid line for Product A
- linestyle="--" → Dashed line for Product B
- color="g" → Green for Product A
- color="r" → Red for Product B

#### Step 4: Add Titles, Labels & Legends

```
plt.title("Monthly Sales Comparison: Product A vs Product B")
plt.xlabel("Months")
plt.ylabel("Sales ($)")
plt.legend()
```

- plt.title() → Adds a title
- plt.xlabel() → Labels the X-axis
- plt.ylabel() → Labels the Y-axis
- plt.legend() → Displays the legend

#### Step 5: Display & Save the Plot

```
plt.savefig("sales_comparison.png") # Save the figure
plt.show() # Display the plot
```

- plt.savefig("sales\_comparison.png") → Saves the figure as PNG
- plt.show() → Displays the line plot

#### Full Code

```
import matplotlib.pyplot as plt

# Data
months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun"]
sales_A = [3000, 4500, 5200, 6100, 7200, 8500] # Sales of Product A
sales_B = [2800, 4000, 4700, 5900, 7100, 8000] # Sales of Product B

# Create the plot
plt.plot(months, sales_A, marker="o", linestyle="-", color="g", label="Product A
Sales")
plt.plot(months, sales_B, marker="s", linestyle="--", color="r", label="Product B
Sales")
```

```
# Add labels, title, and legend
plt.title("Monthly Sales Comparison: Product A vs Product B")
plt.xlabel("Months")
plt.ylabel("Sales ($)")
plt.legend()

# Save and show the plot
plt.savefig("sales_comparison.png")
plt.show()
```

### Expected Output

A line graph comparing Product A and Product B sales, with legends, markers, and different line styles.

## Summary

Feature	Mini Project 1	Mini Project 2
Topic	Website Traffic	Product Sales
Type of Plot	Line Graph	Line Graph
X-axis	Months	Months
Y-axis	Number of Visitors	Sales (\$)
Multiple Lines?	No	Yes
Markers	Yes (circles)	Yes (circles, squares)
Legend?	Yes	Yes
Saved as Image?	Yes (website_traffic.png)	Yes (sales_comparison.png)

These two mini-projects cover all basic Matplotlib concepts, including:

- Plotting simple line graphs
- Adding titles, labels, and legends
- Using multiple line styles and markers
- Saving the figures for reports

## Real-Life Mini Project: Visualizing Sales Performance of a Retail Store

### Objective:

A retail store wants to analyze and visualize its **monthly sales data** using Matplotlib. The store manager needs **clear insights** into sales trends, product performance, and revenue distribution to make better business decisions.

### Day 54 Task

1. What is Matplotlib? Why use it for Data Visualization? – Explain the importance of data visualization in business analytics.
2. Install Matplotlib (pip install matplotlib) – Set up the Matplotlib library in Python.
3. Import Matplotlib (import matplotlib.pyplot as plt) – Load Matplotlib for plotting.
4. Create a Basic Line Plot (plot()) – Visualize monthly total sales using a line graph.
5. Enhance the Plot with Markers & Colors – Use different colors and markers to highlight data points.
6. Add Titles and Labels (title(), xlabel(), ylabel()) – Make the chart more informative with proper labels.
7. Add a Legend (legend()) – Differentiate between sales of different products.
8. Plot a Bar Chart for Category-Wise Sales – Display sales of different product categories using a bar chart.
9. Plot a Pie Chart for Revenue Distribution – Show percentage contribution of different categories in total revenue.

10. Customize the Axes & Grid (grid()) – Improve readability with grid lines and axis limits.
11. Compare Two Product Sales in One Chart – Plot two line graphs (e.g., Electronics vs Clothing) on the same figure.
12. Save the Figure (savefig()) – Store the plot as an image for reports.
13. Display the Final Plot (show()) – Render the graph on the screen for analysis.

## **Real-Life Mini Project Requirements (Updated)**

### **1. Project: Sales Performance Visualization**

Objective: Analyze and visualize monthly sales performance of a store using Matplotlib.

Requirements:

- Load monthly sales data (e.g., revenue for 12 months).
- Create a line plot to show sales trends over time.
- Use different colors to highlight months with the highest and lowest sales.
- Add title, labels, and a legend for clarity.
- Save the final figure as an image (savefig()).

### **2. Project: Social Media Engagement Analysis**

Objective: Visualize user engagement (likes, shares, and comments) across different social media platforms.

Requirements:

- Load engagement data for platforms like Facebook, Instagram, and Twitter.
- Create a bar chart to compare engagement metrics across platforms.

- Use different colors for likes, shares, and comments.
- Add axis labels, title, and a legend to make the visualization clear.
- Save the chart as an image for reporting purposes.

## Day 55

Customizing Plots in Matplotlib (Colors, Styles & Markers)

### Introduction

Matplotlib allows us to customize plots to improve clarity, readability, and aesthetics. Customization includes line styles, markers, grid, figure size, and axis limits.

#### 1. Changing Line Styles

Matplotlib provides different line styles that can be used to modify the appearance of plots.

##### Syntax:

```
plt.plot(x, y, linestyle='--', linewidth=2, color='red')
```

- linestyle: Changes the style of the line (e.g., solid '-', dashed '--', dotted ':').
- linewidth: Adjusts the thickness of the line.
- color: Sets the line color (e.g., 'red', 'blue').

##### Example: Changing Line Style

**Real-Life Scenario:** Suppose we have monthly sales data, and we want to visualize trends with a dashed red line.

### Step-by-Step Implementation

```
import matplotlib.pyplot as plt

# Sample data (Months vs. Sales)
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
sales = [2500, 2700, 2900, 3200, 3100, 3500]

# Plot with customized line style
plt.plot(months, sales, linestyle='--', linewidth=2, color='red')

# Add title and labels
plt.title('Monthly Sales Trend')
plt.xlabel('Months')
plt.ylabel('Sales ($)')

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

**Output:** A red dashed line showing sales trends over 6 months.

## 2. Adding Markers (Highlighting Data Points)

Markers help highlight data points on a line graph.

### Syntax:

```
plt.plot(x, y, marker='o', markersize=8, markerfacecolor='blue')
```

- **marker:** Defines the shape of the marker (e.g., 'o' for circles, '^' for triangles).
- **markersize:** Adjusts marker size.
- **markerfacecolor:** Sets the fill color of markers.

### **Example: Adding Markers**

**Real-Life Scenario:** Suppose we want to highlight sales points with blue circular markers.

#### **Step-by-Step Implementation**

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

```
# Data
```

```
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
sales = [2500, 2700, 2900, 3200, 3100, 3500]
```

```
# Plot with markers
```

```
plt.plot(months, sales, marker='o', markersize=8, markerfacecolor='blue',
         linestyle='-', color='green')
```

```
# Add title and labels
```

```
plt.title('Monthly Sales Trend with Markers')
plt.xlabel('Months')
plt.ylabel('Sales ($)')
```

```
# Show plot
```

```
plt.show()
```

**Output: Green line with blue circular markers at each sales point.**

### **3. Adding Grid (Improving Readability)**

Grids help in reading values easily by providing reference lines.

#### **Syntax:**

```
plt.grid(True, linestyle='--', linewidth=0.5)
```

- True: Enables the grid.
- linestyle: Changes grid line style (e.g., dashed '--').
- linewidth: Sets grid line thickness.

### Example: Adding Grid

**Real-Life Scenario:** To improve readability, let's add a dashed grid to the sales trend plot.

#### Step-by-Step Implementation

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

```
# Data
```

```
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
sales = [2500, 2700, 2900, 3200, 3100, 3500]
```

```
# Plot with grid
```

```
plt.plot(months, sales, marker='o', linestyle='-', color='purple')
plt.grid(True, linestyle='--', linewidth=0.5) # Add a dashed grid
```

```
# Labels and title
```

```
plt.title('Monthly Sales Trend with Grid')
plt.xlabel('Months')
plt.ylabel('Sales ($)')
```

```
# Show plot
```

```
plt.show()
```

**Output:** A sales trend graph with a dashed grid for better readability.

## 4. Changing Figure Size

Adjusting figure size improves visibility when dealing with large datasets.

**Syntax:**

```
plt.figure(figsize=(width, height))
```

- `figsize=(width, height)`: Sets the figure size in inches.

**Example: Changing Figure Size**

**Real-Life Scenario:** If the graph is too small, we can increase its size for better visibility.

**Step-by-Step Implementation**

```
import matplotlib.pyplot as plt

# Increase figure size
plt.figure(figsize=(10, 5))

# Data
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
sales = [2500, 2700, 2900, 3200, 3100, 3500]

# Plot
plt.plot(months, sales, marker='o', linestyle='-', color='orange')

# Labels and title
plt.title('Monthly Sales Trend (Large Size)')
plt.xlabel('Months')
plt.ylabel('Sales ($)')

# Show plot
plt.show()
```

**Output:** A larger plot (10 inches wide, 5 inches tall).

## 5. Adjusting Axis Limits

Setting axis limits helps focus on important data points.

### Syntax:

```
plt.xlim(start, end)
```

```
plt.ylim(start, end)
```

- `xlim(start, end)`: Sets x-axis range.
- `ylim(start, end)`: Sets y-axis range.

### Example: Adjusting Axis Limits

**Real-Life Scenario:** To focus on relevant sales values, let's limit the y-axis to 2000–4000.

#### Step-by-Step Implementation

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

#### # Data

```
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
```

```
sales = [2500, 2700, 2900, 3200, 3100, 3500]
```

#### # Plot

```
plt.plot(months, sales, marker='o', linestyle='-', color='brown')
```

#### # Set axis limits

```
plt.ylim(2000, 4000)
```

#### # Labels and title

```
plt.title('Monthly Sales Trend with Axis Limits')
```

```
plt.xlabel('Months')
```

```
plt.ylabel('Sales ($)')
```

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

**Output:** The y-axis is limited to 2000–4000, making it easier to analyze sales variations.

## Summary Table

Feature	Function Used	Purpose
Change Line Style	linestyle, linewidth, color	Modify appearance of line plots
Add Markers	marker, markersize, markerfacecolor	Highlight data points
Add Grid	grid()	Improve readability
Change Figure Size	figsize=(width, height)	Resize the plot
Adjust Axis Limits	xlim(), ylim()	Focus on important data points

## Conclusion

- ◆ Customizing plots improves clarity, aesthetics, and readability.
- ◆ Matplotlib allows you to modify line styles, add markers, change figure size, add grid lines, and adjust axis limits.
- ◆ These features are useful for business analytics, data visualization, and reports.

## Mini Project 1: Weather Temperature Trends (Customizing Line Styles, Markers & Grid)

### Project Description:

A meteorology department wants to visualize the temperature trends of a city over a week. We will customize the plot using:

- Line styles to differentiate day and night temperatures.
- Markers to highlight specific points.
- Grid lines for better readability.
- Axis limits to keep the focus on relevant temperature values.

### Step 1: Import Required Libraries

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

### Step 2: Define Data

We have day-wise temperatures recorded for morning and night.

```
# Days of the week
days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]

# Temperature readings in Celsius
morning_temp = [22, 24, 23, 25, 26, 27, 28]
night_temp = [18, 19, 20, 21, 22, 23, 22]
```

### Step 3: Create a Customized Line Plot

```
# Set figure size
plt.figure(figsize=(8, 5))

# Plot morning temperature with a solid line and circle markers
plt.plot(days, morning_temp, linestyle='-', linewidth=2, color='red', marker='o',
```

```
markersize=8, markerfacecolor='blue', label="Morning Temperature")  
  
# Plot night temperature with a dashed line and square markers  
plt.plot(days, night_temp, linestyle='--', linewidth=2, color='green', marker='s',  
markersize=8, markerfacecolor='yellow', label="Night Temperature")  
  
# Add title and labels  
plt.title("Weekly Temperature Trends", fontsize=14)  
plt.xlabel("Days", fontsize=12)  
plt.ylabel("Temperature (°C)", fontsize=12)  
  
# Add grid  
plt.grid(True, linestyle='--', linewidth=0.5)  
  
# Add legend  
plt.legend()  
  
# Display the plot  
plt.show()
```

## Output Description

### Customizations Applied:

- Morning temperature is shown with a red solid line and blue circular markers.
- Night temperature is shown with a green dashed line and yellow square markers.
- Grid lines added for better readability.
- Figure size adjusted for better visualization.

**Real-life Use Case:**

This plot can help weather analysts and local citizens understand temperature fluctuations in their city.

**Mini Project 2: Stock Price Movement of a Company****Project Description:**

A financial analyst wants to track a company's stock price movement over 10 days. The analyst needs:

- Different line styles for opening and closing prices.
- Markers to highlight key price points.
- Figure size adjustment for better readability.
- Axis limits to focus on relevant price ranges.

**Step 1: Import Required Libraries**

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

**Step 2: Define Stock Data**

We have daily stock opening and closing prices over 10 days.

```
# Days of tracking
days = ["Day 1", "Day 2", "Day 3", "Day 4", "Day 5", "Day 6", "Day 7", "Day 8",
"Day 9", "Day 10"]
```

```
# Stock prices in USD
opening_price = [100, 102, 101, 105, 110, 108, 107, 111, 113, 115]
closing_price = [102, 101, 104, 107, 109, 107, 108, 112, 114, 116]
```

**Step 3: Create a Customized Stock Price Movement Plot**

```
# Set figure size
plt.figure(figsize=(10, 5))

# Plot opening prices with a solid line and triangle markers
plt.plot(days, opening_price, linestyle='-', linewidth=2, color='blue', marker='^',
         markersize=8, markerfacecolor='red', label="Opening Price")

# Plot closing prices with a dotted line and circle markers
plt.plot(days, closing_price, linestyle=':', linewidth=2, color='purple', marker='o',
         markersize=8, markerfacecolor='orange', label="Closing Price")

# Add title and labels
plt.title("Company Stock Price Movement (10 Days)", fontsize=14)
plt.xlabel("Days", fontsize=12)
plt.ylabel("Stock Price (USD)", fontsize=12)

# Adjust axis limits
plt.ylim(95, 120)

# Add grid
plt.grid(True, linestyle='--', linewidth=0.5)

# Add legend
plt.legend()

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

## Output Description

Customizations Applied:

- Opening price: Blue solid line with red triangle markers.
- Closing price: Purple dotted line with orange circular markers.
- Grid added for better readability.
- Axis limits set to 95 - 120 to keep focus on stock prices.

## Real-life Use Case:

This visualization can help investors, traders, and financial analysts track stock trends over time.

## Summary Table

Customization	Function Used	Purpose
Change Line Style	linestyle, linewidth, color	Modify appearance of line plots
Add Markers	marker, markersize, markerfacecolor	Highlight data points
Add Grid	grid()	Improve readability
Change Figure Size	figsize=(width, height)	Resize the plot
Adjust Axis Limits	xlim(), ylim()	Focus on relevant data

## Conclusion

- ✓ Customizing plots improves readability and presentation in data visualization.
- ✓ These real-life mini projects demonstrate weather analysis and financial analysis.
- ✓ With Matplotlib's customization features, we can create meaningful, clear, and professional-looking visualizations.

## Mini Project: Sales Performance Analysis using Customized Plots

### Project Description:

A retail company wants to analyze and visualize monthly sales trends for three product categories (Electronics, Clothing, and Home Decor) over 12 months. The company needs a customized line plot with:

- Different line styles for each category.
- Markers to highlight key sales points.
- A grid for better readability.
- Figure size adjustment for better clarity.
- Axis limits to focus on relevant sales values.

### Day 55 Tasks

1. Import the necessary libraries (matplotlib.pyplot and numpy if needed).
2. Create a dataset containing monthly sales data for Electronics, Clothing, and Home Decor.
3. Set up the figure size for better visibility.
4. Plot Electronics sales using a solid line style, blue color, and circular markers.
5. Plot Clothing sales using a dashed line style, red color, and square markers.
6. Plot Home Decor sales using a dotted line style, green color, and triangle markers.
7. Customize the markers by adjusting marker size and face color for better clarity.
8. Add a title to the graph to describe the sales analysis.
9. Label the X-axis as "Months" and the Y-axis as "Sales Revenue (in \$1000s)".
10. Add a legend to differentiate the product categories.
11. Add grid lines with a dashed style and light gray color to improve readability.
12. Adjust the Y-axis limits to focus on relevant sales values.
13. Display the final customized plot.

## Expected Outcome:

- A visually appealing line plot that clearly shows monthly sales trends.
- Different styles and markers to distinguish product categories.
- Grid lines and axis adjustments for easy data interpretation.

## Mini Project 1: Daily Water Consumption Analysis

Description: A health and wellness organization wants to analyze daily water consumption of three individuals over a month (30 days). The goal is to visualize the trends and variations in water intake. The plot should include:

- Different line styles and colors for each individual.
- Markers at key consumption points (e.g., highest and lowest intake).
- A grid to enhance readability.
- Figure size adjustments for better visualization.
- Axis limits to focus on a relevant range (e.g., 0-5 liters).

## Mini Project 2: Monthly Electricity Usage of Households

Description: An energy company wants to visualize monthly electricity usage (in kWh) for three households over a 6-month period to identify trends in energy consumption. The visualization should include:

- Distinct line styles and colors for each household.
- Markers at peak and lowest usage points for each household.
- A grid with a subtle color to improve clarity.
- Figure size optimization for clear insights.
- X and Y-axis limits adjustments to focus on the relevant consumption range.

**Bonus Challenge:** Try adding annotations to highlight peak consumption months!  
Let me know if you need further improvements.

# Day 56

## Bar Charts & Histograms in Matplotlib

Data visualization is a crucial part of data analysis. Bar charts and histograms are two of the most common ways to visualize categorical and numerical data. Let's dive deep into their definitions, syntax, and real-life examples with step-by-step implementation.

### Bar Charts

#### Definition

A bar chart is used to represent categorical data with rectangular bars. The length of each bar is proportional to the value it represents.

- Vertical bar chart: `bar()`
- Horizontal bar chart: `barh()`

#### Syntax

```
plt.bar(x, height, color='blue', width=0.8)  
plt.barh(y, width, color='green', height=0.8)
```

- `x`: Categories or labels (e.g., product names, cities)
- `height`: Values for each category (e.g., sales, population)
- `color`: Defines the bar color
- `width`: Adjusts the bar thickness

#### Real-Life Example 1: Sales Data Visualization

A retail store wants to analyze sales for 5 products. We'll use a bar chart to visualize the sales data.

#### Step 1: Import Libraries

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

**Step 2: Create Data**

```
products = ['Laptop', 'Phone', 'Tablet', 'Headphones', 'Smartwatch']
sales = [500, 800, 300, 400, 450]
```

**Step 3: Create Bar Chart**

```
plt.bar(products, sales, color='skyblue', width=0.6)
plt.xlabel("Products")
plt.ylabel("Sales (Units)")
plt.title("Product Sales Report")
plt.show()
```

**Output:** A bar chart displaying the sales of different products.

**Horizontal Bar Chart (barh)**

Sometimes, a horizontal bar chart is better for readability when category labels are long.

**Real-Life Example 2: Population of Cities**

A government agency wants to visualize the population of 4 cities.

```
cities = ['New York', 'Los Angeles', 'Chicago', 'Houston']
population = [8.3, 3.9, 2.7, 2.3] # In millions
```

```
plt.barh(cities, population, color='orange', height=0.6)
plt.xlabel("Population (Millions)")
plt.ylabel("Cities")
plt.title("Population of Major US Cities")
plt.show()
```

**Output:** A horizontal bar chart showing city populations.

## Customizing Bar Colors & Width

We can customize bars by:

- Changing bar colors
- Adjusting bar width
- Using edge colors for better contrast

```
plt.bar(products, sales, color=['red', 'blue', 'green', 'purple', 'cyan'], width=0.5,
edgecolor='black')
plt.xlabel("Products")
plt.ylabel("Sales (Units)")
plt.title("Customized Product Sales Report")
plt.show()
```

**Output:** A bar chart with different colors and black edges.

## Adding Labels & Annotations

To make the chart more informative, we can add value labels on top of bars.

```
plt.bar(products, sales, color='lightblue', width=0.6)

# Adding values on top of bars
for i in range(len(products)):
    plt.text(i, sales[i] + 20, str(sales[i]), ha='center', fontsize=10)
plt.xlabel("Products")
plt.ylabel("Sales (Units)")
plt.title("Sales with Annotations")
plt.show()
```

**Output:** A bar chart with numbers displayed on top of bars.

## Creating a Histogram

### Definition

A histogram is used to represent the distribution of numerical data by dividing it into bins (intervals).

### Syntax

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

- data: Numerical dataset (e.g., student grades, customer ages)
- bins: Number of intervals
- color: Defines histogram bar color
- edgecolor: Defines bar border color

### Real-Life Example 3: Exam Scores Distribution

A school wants to analyze the **distribution of student scores** in a math exam.

#### Step 1: Create Data

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

```
scores = np.random.randint(40, 100, 100) # Generate 100 random scores
```

#### Step 2: Create Histogram

```
plt.hist(scores, bins=10, color='green', edgecolor='black')
plt.xlabel("Scores")
plt.ylabel("Number of Students")
plt.title("Math Exam Scores Distribution")
plt.show()
```

**Output:** A histogram showing the distribution of exam scores.

## Key Takeaways

- Bar charts visualize categorical data (bar(), barh()).
- Customization includes colors, widths, labels, and grid.
- Annotations help in displaying values on top of bars.
- Histograms are useful for numerical data distributions (hist()).
- Adjusting bins can reveal more detailed patterns in data.

## Mini Project 1: Analyzing Monthly Sales Data Using Bar Charts

### Project Requirement

A retail company wants to analyze the monthly sales performance for the year 2024. They need a bar chart to visualize the sales data and compare different months.

### Steps to Implement

#### Step 1: Import Required Libraries

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

```
import numpy as np
```

- matplotlib.pyplot is used to create visualizations.
- numpy will help in generating dummy data.

#### Step 2: Create Monthly Sales Data

```
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
sales = [5000, 7000, 8000, 6000, 9000, 11000, 12000, 10000, 9500, 10500, 9800,
11200]
```

- months: A list of 12 months.
- sales: Corresponding sales values (in dollars) for each month.

**Step 3: Create a Bar Chart**

```
plt.figure(figsize=(10, 5)) # Adjust figure size

plt.bar(months, sales, color='skyblue', edgecolor='black', width=0.6)

plt.xlabel("Months")
plt.ylabel("Sales in $")
plt.title("Monthly Sales Report - 2024")

plt.show()
```

**Output:** A bar chart showing monthly sales trends.

**Step 4: Add Labels & Annotations to Bars**

```
plt.figure(figsize=(10, 5))

plt.bar(months, sales, color='orange', edgecolor='black', width=0.6)

# Adding value labels on top of bars
for i in range(len(months)):
    plt.text(i, sales[i] + 200, f'${sales[i]}', ha='center', fontsize=10)

plt.xlabel("Months")
plt.ylabel("Sales in $")
plt.title("Monthly Sales Report with Labels")

plt.show()
```

**Output:** A bar chart with numerical values displayed on top of bars.

**Step 5: Create a Horizontal Bar Chart**

```
plt.figure(figsize=(10, 5))
plt.barh(months, sales, color='green', edgecolor='black', height=0.6)

plt.xlabel("Sales in $")
plt.ylabel("Months")
plt.title("Monthly Sales Report (Horizontal)")

plt.show()
```

**Output:** A horizontal bar chart for better readability.

**Step 6: Save the Figure**

```
plt.figure(figsize=(10, 5))
plt.bar(months, sales, color='purple', edgecolor='black', width=0.6)

plt.xlabel("Months")
plt.ylabel("Sales in $")
plt.title("Final Monthly Sales Report")

plt.savefig("monthly_sales.png") # Save the plot as an image
plt.show()
```

**Output:** The bar chart will be saved as an image file (monthly\_sales.png).

**Mini Project 2: Analyzing Student Exam Scores Using Histograms****Project Requirement**

A school wants to analyze the exam performance of students in a math test. They have 100 student scores and need a histogram to visualize score distribution.

## Steps to Implement

### Step 1: Import Required Libraries

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

### Step 2: Generate Random Student Scores

```
np.random.seed(42) # Ensure reproducibility
scores = np.random.randint(40, 100, 100) # Generate 100 random scores
```

- np.random.seed(42): Ensures results remain the same every time the code is run.
- np.random.randint(40, 100, 100): Generates 100 random student scores between 40 and 100.

### Step 3: Create a Histogram

```
plt.figure(figsize=(10, 5))
plt.hist(scores, bins=10, color='blue', edgecolor='black')
```

```
plt.xlabel("Exam Scores")
plt.ylabel("Number of Students")
plt.title("Distribution of Math Exam Scores")
```

```
plt.show()
```

**Output:** A histogram showing how student scores are distributed.

### Step 4: Add Customization (Grid & Labels)

```
plt.figure(figsize=(10, 5))
plt.hist(scores, bins=10, color='red', edgecolor='black', alpha=0.7)
```

```
plt.xlabel("Exam Scores")
plt.ylabel("Number of Students")
plt.title("Distribution of Math Exam Scores")

plt.grid(axis='y', linestyle='--', alpha=0.7) # Add grid lines

plt.show()
```

**Output:** A histogram with grid lines for better readability.

#### **Step 5: Adjust Bin Size for More Details**

```
plt.figure(figsize=(10, 5))
plt.hist(scores, bins=15, color='green', edgecolor='black', alpha=0.7)

plt.xlabel("Exam Scores")
plt.ylabel("Number of Students")
plt.title("Detailed Score Distribution (15 Bins)")

plt.show()
```

**Output:** A histogram with 15 bins showing finer details of the score distribution.

#### **Step 6: Save the Histogram**

```
plt.figure(figsize=(10, 5))
plt.hist(scores, bins=10, color='purple', edgecolor='black')

plt.xlabel("Exam Scores")
plt.ylabel("Number of Students")
plt.title("Final Math Score Distribution")
```

```
plt.savefig("exam_scores_distribution.png") # Save as image  
plt.show()
```

**Output:** The histogram will be saved as an image file (exam\_scores\_distribution.png).

## Summary

### Mini Project 1: Monthly Sales Analysis (Bar Chart)

#### Concepts Used:

- bar(), barh()
- Customizing colors, width, edge color
- Adding labels & annotations
- Saving as an image file

### Mini Project 2: Student Exam Scores Analysis (Histogram)

#### Concepts Used:

- hist()
- Customizing bins & colors
- Adding grid lines & labels
- Saving as an image file

## Real-Life Mini Project: Analyzing Movie Box Office Performance

#### Project Description:

A movie production company wants to analyze the box office earnings of different movie genres and the distribution of earnings across multiple movies. They need both bar charts and histograms to visualize the data.

## Day 56 Tasks

1. Create a bar chart showing total earnings for each movie genre using bar().
2. Customize bar colors for better visualization.
3. Adjust the width of bars for better spacing.
4. Add labels for X and Y axes (xlabel(), ylabel()).
5. Add a title to the bar chart (title()).
6. Display the values on top of bars using text() for better readability.
7. Create a horizontal bar chart using barh().
8. Generate random earnings data for 100 different movies.
9. Create a histogram (hist()) to show the distribution of earnings.
10. Set the number of bins to categorize earnings properly.
11. Customize colors and edge color of histogram bars.
12. Add grid lines for better readability using grid().
13. Save both bar chart and histogram as images using savefig().

## Real-Life Mini Project 1: Employee Department-Wise Salary Analysis

### Project Description:

A company wants to analyze the average salaries of employees in different departments and the distribution of salaries across all employees. This project will use bar charts to compare department-wise salaries and histograms to show the overall salary distribution in the company.

## Real-Life Mini Project 2: Website Traffic Analysis

### Project Description:

A digital marketing agency wants to analyze website traffic trends by comparing daily visitors for different pages and visualizing the distribution of session durations. Bar charts will be used to compare traffic for different pages, while histograms will help understand the spread of session durations.

## Day 57

### Scatter Plots & Pie Charts in Matplotlib

Matplotlib provides scatter plots to visualize relationships between two numerical variables and pie charts to show proportions in categorical data. Let's explore them in detail!

#### Scatter Plots

A scatter plot is a type of data visualization that displays points on a two-dimensional coordinate system to show relationships between two variables.

#### Syntax for scatter plot

```
plt.scatter(x, y, color='blue', s=50, label='Category 1')
```

- x, y: The data points to be plotted on the x and y axes.
- color: Sets the color of the points.
- s: Defines the size of the markers.
- label: Adds a label for the legend.

## Real-Life Example: Student Scores vs Study Hours

Scenario: We have data on students' study hours and their scores in an exam. We want to analyze whether more study time improves performance.

### Step 1: Import required libraries

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

```
# Data
```

```
study_hours = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
scores = [40, 50, 55, 65, 70, 75, 78, 85, 88, 92]
```

```
# Scatter plot
```

```
plt.scatter(study_hours, scores, color='red', s=100, label='Student Scores')
```

```
# Customization
```

```
plt.xlabel('Study Hours')
plt.ylabel('Exam Score')
plt.title('Study Hours vs Exam Score')
plt.legend()
plt.grid(True)
```

```
# Show plot
```

```
plt.show()
```

### Explanation:

- study\_hours (x-axis) and scores (y-axis) are plotted.
- color='red', s=100 makes points **more visible**.
- legend(), grid(True) add extra details.

**Conclusion:** The scatter plot may show a positive trend indicating that more study time results in better scores.

## Customizing Scatter Plot Colors & Sizes

You can differentiate data points based on another factor (e.g., gender, category) by modifying color and size dynamically.

```
import numpy as np
```

```
# Example: Age vs Income with size representing work experience
age = np.random.randint(20, 60, 20)
income = np.random.randint(30000, 100000, 20)
experience = np.random.randint(1, 15, 20)

plt.scatter(age, income, c=experience, cmap='viridis', s=experience * 10,
alpha=0.7)
plt.colorbar(label='Years of Experience')
plt.xlabel('Age')
plt.ylabel('Income')
plt.title('Age vs Income with Experience')
plt.show()
```

### Explanation:

- The color (c=experience) and size (s=experience \* 10) vary based on work experience.
- cmap='viridis' applies a gradient color scheme.
- alpha=0.7 makes points partially transparent.

## Adding Labels to Points in Scatter Plot

You can label specific points to highlight important data.

```
students = ['A', 'B', 'C', 'D', 'E']
study_hours = [2, 5, 8, 3, 7]
scores = [50, 70, 90, 55, 85]
```

```
plt.scatter(study_hours, scores, color='blue', s=100)

# Adding labels to points
for i in range(len(study_hours)):
    plt.text(study_hours[i], scores[i], students[i], fontsize=12, ha='right')

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

#### **Explanation:**

- `plt.text(x, y, label)` adds labels to specific data points.

## **Pie Charts**

A pie chart is used to visualize proportions of categories in a dataset.

#### **Syntax for pie chart**

```
plt.pie(values, labels=categories, colors=colors, autopct='%1.1f%%',
explode=explode)
```

- `values`: Data values for each slice.
- `labels`: Names of the categories.
- `colors`: Custom colors for each section.
- `autopct='%1.1f%%'`: Shows percentage values.
- `explode`: Moves certain slices out to highlight them.

## Real-Life Example: Market Share of Companies

**Scenario:** A company wants to analyze the market share of different smartphone brands.

### Step 1: Import and prepare data

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

```
# Data
```

```
brands = ['Apple', 'Samsung', 'Xiaomi', 'OnePlus', 'Others']
market_share = [35, 30, 15, 10, 10]
colors = ['gold', 'blue', 'red', 'green', 'gray']
explode = [0.1, 0, 0, 0, 0] # Highlight Apple
```

```
# Creating Pie Chart
```

```
plt.pie(market_share, labels=brands, colors=colors, autopct='%.1f%%',
explode=explode, startangle=140)
```

```
# Adding title
```

```
plt.title('Smartphone Market Share')
```

```
# Show plot
```

```
plt.show()
```

### Explanation:

- The largest market share (Apple) is highlighted using `explode=[0.1, 0, 0, 0, 0]`.
- `autopct='%.1f%%'` displays percentage values.

## Customizing Pie Charts

You can change colors, add labels, and explode slices for emphasis.

```
# Custom Pie Chart for Social Media Usage
platforms = ['Facebook', 'Instagram', 'Twitter', 'TikTok', 'Snapchat']
users = [40, 30, 10, 15, 5]
colors = ['blue', 'purple', 'lightblue', 'red', 'yellow']
explode = [0, 0.1, 0, 0, 0] # Highlight Instagram

plt.pie(users, labels=platforms, colors=colors, autopct='%1.1f%%',
explode=explode, startangle=90, shadow=True)
plt.title('Social Media Platform Usage')
plt.show()
```

**Explanation:**

- `explode=[0, 0.1, 0, 0, 0]` highlights Instagram.
- `shadow=True` adds a 3D effect.

**Key Takeaways**

- ✓ Scatter plots show relationships between two numerical variables.
- ✓ Pie charts display proportions in categorical data.
- ✓ You can customize scatter plots with color, size, labels.
- ✓ Pie charts can be exploded, colored, and labeled for better visibility.

**Mini Project 1: Analyzing Employee Salary vs Experience using Scatter Plot****Project Overview:**

A company wants to analyze whether years of experience impact an employee's salary. You will create a scatter plot to visualize this relationship and customize it using colors, sizes, and labels.

**Step 1: Import Required Libraries**

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

**Step 2: Prepare Sample Data**

```
# Years of experience (x-axis)  
experience = np.array([1, 2, 3, 5, 7, 8, 10, 12, 15, 18])  
  
# Corresponding salaries in thousands (y-axis)  
salaries = np.array([30, 35, 40, 50, 65, 75, 85, 100, 120, 140])  
  
# Sizes of markers (representing importance or seniority)  
marker_size = np.array([50, 70, 90, 110, 130, 150, 170, 200, 220, 250])
```

**Step 3: Create a Scatter Plot with Customization**

```
plt.figure(figsize=(8, 5)) # Set figure size  
  
# Scatter plot with color, marker size, and labels  
plt.scatter(experience, salaries, color='blue', s=marker_size, alpha=0.6,  
edgecolors='black', label='Employees')  
  
# Adding labels for each point  
for i in range(len(experience)):  
    plt.text(experience[i], salaries[i], f"{salaries[i}]K", fontsize=10, ha='right')  
  
# Customizing the chart  
plt.xlabel('Years of Experience')  
plt.ylabel('Salary (in 1000s)')  
plt.title('Employee Salary vs Experience')  
plt.legend()
```

```
plt.grid(True)
```

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

### **Explanation**

- ✓ Experience (x-axis) vs Salary (y-axis) is plotted.
- ✓ color='blue' makes the points visible, alpha=0.6 adds transparency.
- ✓ marker\_size differentiates seniority levels of employees.
- ✓ plt.text() adds salary labels to each point.
- ✓ grid(True) makes the chart readable.

**Conclusion:** The scatter plot helps HR analyze salary trends, revealing if more experience results in higher salaries.

## **Mini Project 2: Market Share of Car Brands using a Pie Chart**

### **Project Overview:**

An automobile company wants to analyze the market share of different car brands. You will use a pie chart to visualize brand dominance.

### **Step 1: Import Required Libraries**

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

### **Step 2: Define Data for Car Market Share**

```
# Car brands and their market share percentages
brands = ['Toyota', 'Ford', 'Honda', 'BMW', 'Tesla', 'Others']
market_share = [25, 20, 18, 15, 12, 10]
```

```
# Define custom colors for each brand  
colors = ['gold', 'blue', 'red', 'green', 'purple', 'gray']  
  
# Exploding the most dominant brand (Toyota)  
explode = [0.1, 0, 0, 0, 0, 0]
```

### Step 3: Create a Pie Chart with Customization

```
plt.figure(figsize=(8, 6)) # Set figure size  
  
# Create pie chart  
plt.pie(market_share, labels=brands, colors=colors, autopct='%.1f%%',  
        explode=explode, startangle=140, shadow=True)  
  
# Add title  
plt.title('Market Share of Car Brands in 2024')  
  
# Display plot  
plt.show()
```

### Explanation

- ✓ Car brands and their respective market shares are visualized.
- ✓ `explode=[0.1, 0, 0, 0, 0, 0]` highlights Toyota as the leading brand.
- ✓ `autopct='%.1f%%'` adds percentage values for clarity.
- ✓ `shadow=True` adds a 3D effect.

**Conclusion:** The pie chart clearly shows market share distribution, helping businesses understand competitors.

## Real-Life Mini Project: Analyzing Sales Performance using Scatter & Pie Charts

### Project Overview:

A retail company wants to analyze sales performance by visualizing:

1. Product Sales vs Profit using a scatter plot
2. Sales Contribution of Different Regions using a pie chart

Your tasks will involve creating, customizing, and labeling scatter plots and pie charts for better insights.

### Day 57 Tasks

#### Task 1: Load Required Libraries

- Import matplotlib.pyplot and numpy for data visualization.

#### Task 2: Create Sample Sales Data for Scatter Plot

- Define sales revenue and corresponding profit margins for multiple products.

#### Task 3: Create a Basic Scatter Plot using scatter()

- Plot sales revenue (x-axis) vs profit margin (y-axis).

#### Task 4: Customize Scatter Plot Colors & Sizes

- Use different colors for high vs low profits.
- Adjust marker size based on total units sold.

#### Task 5: Add Labels to Scatter Points

- Label each data point with the product name for clarity.

#### Task 6: Improve Readability with Titles & Grid

- Add a title, x-label, and y-label.
- Enable a grid for better visualization.

#### Task 7: Create Sales Data for Pie Chart

- Define sales contribution of different regions (North, South, East, West).

#### Task 8: Create a Basic Pie Chart using pie()

- Visualize sales distribution across regions.

#### Task 9: Customize Pie Chart Colors

- Assign a unique color to each region.

#### Task 10: Highlight the Best-Performing Region (Explode Effect)

- Use the explode parameter to highlight the region with the highest sales.

#### Task 11: Add Percentage Labels to Pie Chart

- Display percentage values inside the pie slices.

#### Task 12: Improve Readability with Titles & Shadows

- Add a title and apply a shadow effect for a professional look.

#### Task 13: Display Both Charts Together

- Use subplots to show both scatter plot & pie chart in a single figure.

#### **End Goal:**

- ✓ A scatter plot showing product sales vs profit to analyze trends.
- ✓ A pie chart visualizing regional sales contribution for better decision-making.

## Mini Project 1: Customer Age vs Spending Behavior Analysis

### Objective:

A shopping mall wants to analyze customer spending behavior based on their age. You need to:

1. Create a scatter plot to visualize the relationship between customer age and average monthly spending.
2. Customize colors and sizes to differentiate between spending categories (low, medium, high).
3. Label important data points such as high spenders.
4. Create a pie chart showing the percentage of customers in different spending categories.
5. Customize the pie chart by adding colors, labels, and highlighting the most frequent spending category.

## Mini Project 2: Employee Distribution & Salary Analysis

### Objective:

A company wants to visualize employee distribution across departments and analyze salaries. You need to:

1. Create a scatter plot to show the relationship between years of experience and salary.
2. Use different colors and sizes to represent different job levels (Junior, Mid, Senior).
3. Add labels to key employees (e.g., highest paid, lowest experience).
4. Create a pie chart showing the percentage of employees in each department (IT, HR, Sales, Marketing).

5. Customize the pie chart by highlighting the largest department, adding labels, and applying a shadow effect.

## Day 58

### Subplots & Multiple Plots in Matplotlib

#### Introduction

When working with data visualization, you may need to display multiple plots in a single figure to compare datasets efficiently. Matplotlib provides powerful functions to create subplots, adjust layouts, and share axes for better organization and readability.

#### 1. Creating Multiple Plots using subplot()

##### Definition

The subplot() function allows you to create multiple plots in a single figure by specifying the number of rows, columns, and the index of the current plot.

##### Syntax

```
plt.subplot(nrows, ncols, index)
```

- nrows: Number of rows in the grid.
- ncols: Number of columns in the grid.
- index: The position of the current plot (starting from 1, left to right, top to bottom).

## Example

Creating 2 Plots in One Figure

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

# Sample data
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# Creating subplots
plt.subplot(2, 1, 1) # (2 rows, 1 column, first plot)
plt.plot(x, y1, 'r', label='Sine Wave')
plt.title("Sine & Cosine Waves")
plt.legend()

plt.subplot(2, 1, 2) # (2 rows, 1 column, second plot)
plt.plot(x, y2, 'b', label='Cosine Wave')
plt.legend()

plt.show()
```

## Output

Two vertically arranged plots: Sine wave (top) and Cosine wave (bottom).

## 2. Arranging Plots in Rows & Columns

Instead of placing plots in a single row or column, you can arrange them in a grid format.

**Example: Creating a 2x2 Grid of Plots**

```
plt.figure(figsize=(8,6)) # Set figure size

plt.subplot(2,2,1) # First subplot
plt.plot(x, y1, 'r')
plt.title('Plot 1')

plt.subplot(2,2,2) # Second subplot
plt.plot(x, y2, 'b')
plt.title('Plot 2')

plt.subplot(2,2,3) # Third subplot
plt.plot(x, -y1, 'g')
plt.title('Plot 3')

plt.subplot(2,2,4) # Fourth subplot
plt.plot(x, -y2, 'm')
plt.title('Plot 4')

plt.show()
```

**Output**

A 2x2 grid of subplots displaying different variations of sine and cosine waves.

**3. Adjusting Layout with `tight_layout()`****Definition**

`plt.tight_layout()` automatically adjusts subplot spacing to prevent overlapping labels.

**Example**

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

```
plt.subplot(2,2,1)
plt.plot(x, y1, 'r')
plt.title('Plot 1')
```

```
plt.subplot(2,2,2)
plt.plot(x, y2, 'b')
plt.title('Plot 2')
```

```
plt.subplot(2,2,3)
plt.plot(x, -y1, 'g')
plt.title('Plot 3')
```

```
plt.subplot(2,2,4)
plt.plot(x, -y2, 'm')
plt.title('Plot 4')
```

```
plt.tight_layout() # Adjusts spacing
plt.show()
```

**Output**

Same 2×2 subplot layout without overlapping labels.

## 4. Sharing Axes in Subplots (`sharex`, `sharey`)

When comparing multiple datasets, sharing X or Y axes ensures alignment.

**Syntax**

```
fig, axes = plt.subplots(nrows, ncols, sharex=True, sharey=True)
```

- sharex=True: Shares the X-axis.
- sharey=True: Shares the Y-axis.

### **Example**

```
fig, axes = plt.subplots(2, 2, sharex=True, sharey=True, figsize=(8,6))
axes[0,0].plot(x, y1, 'r')
axes[0,0].set_title("Plot 1")

axes[0,1].plot(x, y2, 'b')
axes[0,1].set_title("Plot 2")

axes[1,0].plot(x, -y1, 'g')
axes[1,0].set_title("Plot 3")

axes[1,1].plot(x, -y2, 'm')
axes[1,1].set_title("Plot 4")

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

### **Output**

A  $2 \times 2$  subplot layout where all plots share the same X and Y axes.

## **5. Using plt.subplots() for Advanced Customization**

`plt.subplots()` is a more flexible alternative to `subplot()`, allowing better subplot control.

### **Syntax**

```
fig, axes = plt.subplots(nrows, ncols, figsize=(width, height))
```

- fig: The figure object.
- axes: An array of subplot objects.

### Example

```

fig, axes = plt.subplots(2, 2, figsize=(8,6))
axes[0,0].plot(x, y1, 'r', label="Sine")
axes[0,0].legend()

axes[0,1].plot(x, y2, 'b', label="Cosine")
axes[0,1].legend()

axes[1,0].plot(x, -y1, 'g', label="Negative Sine")
axes[1,0].legend()

axes[1,1].plot(x, -y2, 'm', label="Negative Cosine")
axes[1,1].legend()

plt.tight_layout()
plt.show()

```

### Output

A fully customizable 2x2 subplot layout with legends and better spacing.

### Conclusion

Feature	Method
Create Multiple Plots	plt.subplot()
Arrange in Rows & Columns	plt.subplot(rows, cols, index)
Adjust Layout	plt.tight_layout()
Share Axes	sharex=True, sharey=True
Advanced Customization	plt.subplots()

By combining these techniques, you can compare multiple datasets efficiently in a well-structured layout.

## Mini Project 1: Analyzing Weather Trends with Subplots

### Problem Statement

A weather analysis team wants to visualize temperature, humidity, wind speed, and rainfall trends over time. They need a single figure with multiple subplots to compare these weather parameters efficiently.

### Steps to Implement

1. Import Required Libraries
2. Create Sample Weather Data (Days, Temperature, Humidity, Wind Speed, Rainfall)
3. Create a Figure with Multiple Subplots
  - a. Use plt.subplots() for better customization.
  - b. Arrange plots in a 2x2 grid.
4. Plot Temperature Data (Line graph)
5. Plot Humidity Data (Bar chart)
6. Plot Wind Speed Data (Scatter plot)
7. Plot Rainfall Data (Histogram)
8. Customize Subplots
  - a. Titles, axis labels, legends.
  - b. Adjust layout using tight\_layout().
9. Show the Final Figure

### Step-by-Step Implementation with Code Explanation

```
import matplotlib.pyplot as plt
import numpy as np
# Step 1: Generate Sample Data (7 Days)
days = np.arange(1, 8)
temperature = [30, 32, 31, 29, 28, 27, 26] # in Celsius
humidity = [60, 65, 58, 55, 50, 48, 47] # in percentage
wind_speed = [10, 12, 9, 11, 14, 13, 15] # in km/h
rainfall = [5, 20, 15, 30, 10, 25, 18] # in mm
```

```
# Step 2: Create a Figure with 2x2 Subplots
fig, axes = plt.subplots(2, 2, figsize=(10, 6))

# Step 3: Temperature Plot (Line Graph)
axes[0, 0].plot(days, temperature, 'r-o', label="Temperature (°C)")
axes[0, 0].set_title("Temperature Over a Week")
axes[0, 0].set_xlabel("Days")
axes[0, 0].set_ylabel("Temperature (°C)")
axes[0, 0].legend()
axes[0, 0].grid(True)

# Step 4: Humidity Plot (Bar Chart)
axes[0, 1].bar(days, humidity, color='b', label="Humidity (%)")
axes[0, 1].set_title("Humidity Levels Over a Week")
axes[0, 1].set_xlabel("Days")
axes[0, 1].set_ylabel("Humidity (%)")
axes[0, 1].legend()

# Step 5: Wind Speed Plot (Scatter Plot)
axes[1, 0].scatter(days, wind_speed, color='g', label="Wind Speed (km/h)")
axes[1, 0].set_title("Wind Speed Over a Week")
axes[1, 0].set_xlabel("Days")
axes[1, 0].set_ylabel("Wind Speed (km/h)")
axes[1, 0].legend()

# Step 6: Rainfall Plot (Histogram)
axes[1, 1].hist(rainfall, bins=5, color='purple', edgecolor='black', label="Rainfall (mm)")
axes[1, 1].set_title("Rainfall Distribution")
axes[1, 1].set_xlabel("Rainfall (mm)")
axes[1, 1].set_ylabel("Frequency")
```

```

axes[1, 1].legend()

# Step 7: Adjust Layout & Display
plt.tight_layout()
plt.show()

```

### **Expected Output**

A 2x2 grid of plots showing:

- Line chart for temperature trends.
- Bar chart for humidity levels.
- Scatter plot for wind speed variations.
- Histogram for rainfall distribution.

## **Mini Project 2: Stock Market Performance Analysis**

### **Problem Statement**

A financial analyst wants to compare stock performance for four major companies over a period of time. Using multiple plots in a single figure, they can track stock price fluctuations and trends.

### **Steps to Implement**

1. Import Required Libraries
2. Generate Sample Stock Price Data for four companies over 10 days.
3. Create a Figure with Multiple Subplots
  - a. Use plt.subplots() with sharex=True for aligned time.
4. Plot Stock Prices for Each Company (Using different colors & styles)
5. Customize Each Plot
  - a. Add grid, title, and labels.
6. Optimize Layout using tight\_layout()
7. Display the Final Figure

**Step-by-Step Implementation with Code Explanation**

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

# Step 1: Generate Sample Stock Price Data (10 Days)
days = np.arange(1, 11)
company_A = [150, 152, 148, 155, 160, 158, 162, 165, 168, 170]
company_B = [220, 215, 225, 230, 228, 235, 240, 245, 250, 255]
company_C = [310, 305, 315, 320, 318, 325, 330, 335, 340, 345]
company_D = [90, 92, 95, 98, 97, 100, 102, 105, 108, 110]

# Step 2: Create Subplots (4 Rows, 1 Column) with Shared X-Axis
fig, axes = plt.subplots(4, 1, figsize=(8, 10), sharex=True)

# Step 3: Plot Company A Stock Prices
axes[0].plot(days, company_A, 'r-o', label="Company A")
axes[0].set_title("Company A Stock Prices")
axes[0].set_ylabel("Price ($)")
axes[0].legend()
axes[0].grid(True)

# Step 4: Plot Company B Stock Prices
axes[1].plot(days, company_B, 'b-s', label="Company B")
axes[1].set_title("Company B Stock Prices")
axes[1].set_ylabel("Price ($)")
axes[1].legend()
axes[1].grid(True)

# Step 5: Plot Company C Stock Prices
axes[2].plot(days, company_C, 'g-^', label="Company C")
axes[2].set_title("Company C Stock Prices")
```

```

axes[2].set_ylabel("Price ($)")
axes[2].legend()
axes[2].grid(True)

# Step 6: Plot Company D Stock Prices
axes[3].plot(days, company_D, 'm-d', label="Company D")
axes[3].set_title("Company D Stock Prices")
axes[3].set_xlabel("Days")
axes[3].set_ylabel("Price ($)")
axes[3].legend()
axes[3].grid(True)

# Step 7: Adjust Layout & Show Plot
plt.tight_layout()
plt.show()

```

### Expected Output

A 4-row subplot visualization of stock price trends for four companies over 10 days, sharing the same X-axis for easy comparison.

### Key Takeaways

Feature	Mini Project 1 (Weather)	Mini Project 2 (Stock Market)
Data Type	Temperature, Humidity, Wind, Rainfall	Stock Prices of 4 Companies
Subplot Layout	2x2 Grid	4 Rows, 1 Column
Plot Types	Line, Bar, Scatter, Histogram	Line Graphs
X-Axis Sharing	No	Yes (Stock Prices over Days)
Customization	Legends, Grid, Titles	Legends, Grid, Titles

These projects help visualize multiple datasets efficiently for real-world analysis.

## Real-Life Mini Project: Analyzing Air Pollution Levels Across Cities

### Objective:

A government environmental agency wants to analyze air pollution levels (PM2.5, PM10, CO, NO2) in different cities using multiple subplots to compare trends effectively.

### Day 58 Tasks

1. Import Necessary Libraries (Matplotlib, NumPy, Pandas for data handling).
2. Create Sample Data for air pollution levels in 4 major cities over a period of 7 days.
3. Initialize a Figure with Subplots (2 rows × 2 columns layout).
4. Plot PM2.5 Levels (Line chart for each city).
5. Plot PM10 Levels (Bar chart for each city).
6. Plot CO Concentration (Scatter plot with color variation).
7. Plot NO2 Concentration (Histogram to show frequency distribution).
8. Share X-Axis for Time Series Data (Use sharex=True for better alignment).
9. Customize Each Plot (Titles, X & Y labels, grid, legends).
10. Adjust Figure Size to ensure readability.
11. Use `tight_layout()` to improve spacing between subplots.
12. Save the Final Figure as an image for reports.
13. Display the Final Visualization showing pollution trends across cities.

This mini-project will help visualize air pollution trends effectively, aiding in better decision-making and policy planning.

## Real-Life Mini Project 1: Sales Performance Analysis of Different Product Categories

### Objective:

A retail company wants to analyze sales trends across different product categories (Electronics, Clothing, Home Appliances, and Books) over a period of 12 months using multiple subplots.

## Real-Life Mini Project 2: Weather Trends Comparison Across Cities

### Objective:

A meteorological department wants to compare weather parameters (Temperature, Humidity, Rainfall, and Wind Speed) across four different cities over a year using multiple plots in a structured layout.

# Day 59

## Advanced Customizations & 3D Plots in Matplotlib

Matplotlib provides powerful customization features to enhance data visualization. With annotations, 3D plotting, and heatmaps, we can create insightful representations of data.

### 1. Adding Annotations & Text to Plots (annotate())

#### Definition:

Annotations are used to highlight important points in a plot by adding text or arrows. This helps in making plots more readable and informative.

**Syntax:**

```
plt.annotate(text, xy, xytext, arrowprops)
```

- text: The text to display.
- xy: The point where the annotation is attached.
- xytext: The location where the text should appear.
- arrowprops: A dictionary specifying arrow properties.

**Real-Life Example: Highlighting Peak Sales Month**

**Scenario:** A store wants to highlight the month with the highest sales in a line chart.

**Step-by-Step Implementation:**

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

```
# Data
```

```
months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun"]
```

```
sales = [100, 150, 180, 200, 300, 250]
```

```
# Plot
```

```
plt.plot(months, sales, marker='o', linestyle='-', color='b', label="Sales")
```

```
# Annotate highest sales point
```

```
plt.annotate("Peak Sales", xy=("May", 300), xytext=("Mar", 320),
            arrowprops=dict(facecolor='red', shrink=0.05), fontsize=12, color='red')
```

```
# Labels
```

```
plt.xlabel("Months")
```

```
plt.ylabel("Sales ($)")
```

```
plt.title("Monthly Sales Data")
```

```
plt.legend()
```

```
# Show Plot
plt.show()
```

**Explanation:**

- The peak sales month ("May") is highlighted with an annotation.
- An arrow points to the highest sales value (300).

## 2. Creating 3D Plots using mpl\_toolkits.mplot3d

**Definition:**

Matplotlib allows creating 3D plots using `mpl_toolkits.mplot3d`, useful for visualizing data with three variables (X, Y, and Z).

**Syntax:**

```
from mpl_toolkits.mplot3d import Axes3D
ax = fig.add_subplot(111, projection='3d')
```

- `fig.add_subplot(111, projection='3d')`: Creates a 3D plot.

**Real-Life Example: Visualizing Mountain Heights**

**Scenario:** A geographer wants to visualize the heights of mountains across different regions.

**Step-by-Step Implementation:**

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np

# Data
x = np.array([1, 2, 3, 4, 5]) # Regions
y = np.array([10, 20, 30, 40, 50]) # Distance from base
```

```
z = np.array([500, 700, 1000, 1200, 1500]) # Height in meters

# Create Figure
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# 3D Line Plot
ax.plot(x, y, z, marker='o', linestyle='-', color='g')

# Labels
ax.set_xlabel("Regions")
ax.set_ylabel("Distance")
ax.set_zlabel("Height (m)")
ax.set_title("Mountain Heights Across Regions")

# Show Plot
plt.show()
```

**Explanation:**

- The plot3D() function creates a 3D line showing mountain heights.
- Labels are added for better readability.

### 3. Customizing 3D Plots (plot3D(), scatter3D(), bar3D())

**Definition:**

Matplotlib allows multiple types of 3D visualizations:

- plot3D(): Creates 3D line plots.
- scatter3D(): Creates 3D scatter plots.
- bar3D(): Creates 3D bar charts.

## Real-Life Example: Visualizing City Population Density

**Scenario:** A government agency wants to analyze the population density of cities.

Step-by-Step Implementation:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Data
cities = ["City A", "City B", "City C", "City D"]
x = np.arange(len(cities))
y = np.array([100, 200, 300, 400]) # Area in square km
z = np.array([500000, 700000, 1000000, 1200000]) # Population

# Create Figure
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# 3D Bar Chart
ax.bar3d(x, y, np.zeros_like(z), dx=0.5, dy=50, dz=z, color='b')

# Labels
ax.set_xticks(x)
ax.set_xticklabels(cities)
ax.set_xlabel("Cities")
ax.set_ylabel("Area (sq km)")
ax.set_zlabel("Population")
ax.set_title("City Population Density")

# Show Plot
plt.show()
```

**Explanation:**

- `bar3D()` creates 3D bars representing population in different cities.
- `set_xticks()` assigns city names to X-axis.

**4. Creating Heatmaps with Matplotlib****Definition:**

A heatmap is a graphical representation of data where values are shown using different colors.

**Syntax:**

```
plt.imshow(data, cmap='coolwarm', interpolation='nearest')
```

- `data`: The 2D array of values to visualize.
- `cmap`: Defines the color scheme (e.g., `coolwarm`, `viridis`).
- `interpolation`: Specifies how colors are displayed (e.g., `nearest`).

**Real-Life Example: Analyzing Monthly Temperature Variation**

**Scenario:** A weather station wants to visualize the temperature changes over a year in different cities.

**Step-by-Step Implementation:**

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

```
# Data: Monthly temperatures (rows: months, columns: cities)
data = np.array([
    [30, 32, 29, 31],
    [28, 29, 27, 30],
    [26, 28, 25, 27],
    [24, 26, 23, 25]
])
```

```
# Create Heatmap
plt.imshow(data, cmap='coolwarm', interpolation='nearest')

# Labels
plt.colorbar(label="Temperature (°C)")
plt.xticks(ticks=[0, 1, 2, 3], labels=["City A", "City B", "City C", "City D"])
plt.yticks(ticks=[0, 1, 2, 3], labels=["Jan", "Feb", "Mar", "Apr"])
plt.title("Temperature Variation Across Cities")

# Show Plot
plt.show()
```

**Explanation:**

- The heatmap represents temperature variations across different cities.
- colorbar() adds a color legend for temperature values.

**Conclusion**

Feature	Usage
annotate()	Highlights key data points with text and arrows.
plot3D(), scatter3D(), bar3D()	Creates various 3D visualizations.
bar3D()	Visualizes comparisons in 3D bars.
imshow()	Generates a heatmap for visualizing intensity data.

These techniques help in enhancing data insights through effective visualization.

## Mini Project 1: Analyzing and Visualizing Air Pollution Levels in Cities using 3D Plots and Heatmaps

### Project Overview:

Air pollution is a significant environmental issue affecting many cities. In this project, we will analyze air pollution levels across different cities using 3D plots and heatmaps.

### Key Features:

- ✓ 3D Scatter Plot to visualize pollution levels based on three pollutants (PM2.5, NO<sub>2</sub>, and CO).
- ✓ 3D Bar Chart to compare pollution levels in different cities.
- ✓ Heatmap to show monthly pollution trends.
- ✓ Annotations to highlight the most and least polluted cities.

### Step 1: Import Required Libraries

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

### Step 2: Prepare Air Pollution Data

We consider four cities and three pollutants (PM2.5, NO<sub>2</sub>, CO).

```
# City names
cities = ["New York", "Los Angeles", "Chicago", "Houston"]

# Pollution Data (PM2.5, NO2, CO in µg/m3)
pollution_levels = np.array([
    [35, 50, 7], # New York
```

```
[55, 70, 10], # Los Angeles  
[40, 60, 8], # Chicago  
[30, 45, 6] # Houston  
])
```

```
# Separate pollution levels  
pm25 = pollution_levels[:, 0]  
no2 = pollution_levels[:, 1]  
co = pollution_levels[:, 2]
```

### Step 3: Create a 3D Scatter Plot for Pollution Levels

```
# Create figure  
fig = plt.figure(figsize=(8, 6))  
ax = fig.add_subplot(111, projection='3d')  
  
# Scatter plot  
ax.scatter(pm25, no2, co, color='red', marker='o')  
  
# Labels  
ax.set_xlabel("PM2.5 ( $\mu\text{g}/\text{m}^3$ )")  
ax.set_ylabel("NO2 ( $\mu\text{g}/\text{m}^3$ )")  
ax.set_zlabel("CO ( $\mu\text{g}/\text{m}^3$ )")  
ax.set_title("3D Scatter Plot of Air Pollution Levels")  
  
# Annotate most polluted city  
ax.text(55, 70, 10, "Los Angeles", color='blue', fontsize=10)  
  
# Show plot  
plt.show()
```

**Explanation:**

- scatter() creates a 3D scatter plot of pollution levels.
- The highest polluted city (Los Angeles) is highlighted using text().

**Step 4: Create a 3D Bar Chart to Compare Cities**

```

fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
# X-axis positions
x_pos = np.arange(len(cities))
y_pos = np.zeros_like(x_pos)

# Bar chart for each pollutant
ax.bar3d(x_pos, y_pos, np.zeros_like(pm25), dx=0.5, dy=0.5, dz=pm25, color='r',
label="PM2.5")
ax.bar3d(x_pos, y_pos+1, np.zeros_like(no2), dx=0.5, dy=0.5, dz=no2, color='g',
label="NO2")
ax.bar3d(x_pos, y_pos+2, np.zeros_like(co), dx=0.5, dy=0.5, dz=co, color='b',
label="CO")

# Labels
ax.set_xticks(x_pos)
ax.set_xticklabels(cities)
ax.set_xlabel("Cities")
ax.set_ylabel("Pollutants")
ax.set_zlabel("Concentration ( $\mu\text{g}/\text{m}^3$ )")
ax.set_title("3D Bar Chart of Air Pollution")

# Show plot
plt.legend()
plt.show()

```

**Explanation:**

- `bar3d()` is used to visualize pollution levels in different cities.
- The three pollutants (PM2.5, NO2, CO) are represented as separate bars.

**Step 5: Create a Heatmap of Monthly Pollution Trends**

```
# Monthly pollution data (rows: months, columns: cities)
monthly_pollution = np.random.randint(20, 80, size=(12, 4))

# Create Heatmap
plt.figure(figsize=(8, 5))
plt.imshow(monthly_pollution, cmap='coolwarm', interpolation='nearest')

# Add color bar
plt.colorbar(label="Pollution Level ( $\mu\text{g}/\text{m}^3$ )")

# Labels
plt.xticks(ticks=np.arange(len(cities)), labels=cities)
plt.yticks(ticks=np.arange(12), labels=["Jan", "Feb", "Mar", "Apr", "May", "Jun",
"Jul", "Aug", "Sep", "Oct", "Nov", "Dec"])
plt.title("Heatmap of Monthly Pollution Trends")

# Show plot
plt.show()
```

**Explanation:**

- `imshow()` generates a heatmap showing pollution trends over 12 months.
- The color intensity represents pollution levels.

## Mini Project 2: Visualizing Sales Performance Across Stores Using 3D Plots & Annotations

### Project Overview:

A company wants to analyze the sales performance of different stores using 3D plots, bar charts, and annotations.

### Key Features:

- ✓ 3D Line Plot to analyze sales trends.
- ✓ 3D Bar Chart to compare sales in different stores.
- ✓ Annotations to highlight best and worst performers.

### Step 1: Import Required Libraries

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

### Step 2: Define Sales Data

```
# Store names
stores = ["Store A", "Store B", "Store C", "Store D"]
# Sales Data (X: Month, Y: Store, Z: Sales in $1000)
months = np.arange(1, 7) # 6 months
sales = np.array([
    [30, 40, 35, 50],
    [40, 45, 42, 55],
    [50, 55, 48, 60],
    [60, 65, 55, 65],
    [70, 75, 65, 70],
    [80, 85, 75, 75]
])
```

**Step 3: Create a 3D Line Plot for Sales Trends**

```
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')

# Plot each store's sales over time
for i in range(len(stores)):
    ax.plot(months, np.full_like(months, i), sales[:, i], marker='o', linestyle='-', label=stores[i])

# Labels
ax.set_xlabel("Months")
ax.set_ylabel("Stores")
ax.set_zlabel("Sales ($1000)")
ax.set_title("3D Line Plot of Store Sales Performance")
ax.set_yticks(np.arange(len(stores)))
ax.set_yticklabels(stores)

# Highlight best and worst store
ax.text(6, 3, 75, "Best Store", color='green', fontsize=10)
ax.text(1, 0, 30, "Lowest Sales", color='red', fontsize=10)

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

**Explanation:**

- `plot()` generates a 3D sales trend line for each store.
- The best and worst-performing stores are highlighted with annotations.

**Step 4: Create a 3D Bar Chart to Compare Stores**

```
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')

# Bar chart
x_pos = np.arange(len(stores))
ax.bar3d(x_pos, np.zeros_like(x_pos), np.zeros_like(x_pos), dx=0.5, dy=0.5,
dz=sales[-1], color='blue')

# Labels
ax.set_xticks(x_pos)
ax.set_xticklabels(stores)
ax.set_xlabel("Stores")
ax.set_ylabel("Months")
ax.set_zlabel("Sales ($1000)")
ax.set_title("3D Bar Chart of Sales Performance")

# Show plot
plt.show()
```

**Explanation:**

- The bar height represents sales in the last month.
- It helps in comparing store performances.

**Conclusion**

These projects demonstrate how 3D plots, annotations, and heatmaps can be used in real-world business and environmental data visualization.

## Real-Life Mini Project: Weather Data Visualization Using 3D Plots & Heatmaps

### Project Overview:

In this project, you will visualize weather data (temperature, humidity, and wind speed) of different cities using 3D plots, scatter plots, bar charts, and heatmaps.

### Day 59 Tasks

#### 1. Load & Prepare Weather Data

- Load weather data containing temperature, humidity, and wind speed for different cities.
- Convert the data into NumPy arrays for easy manipulation.

#### 2. Create a 3D Scatter Plot of Weather Data

- Plot temperature, humidity, and wind speed in a 3D scatter plot using scatter3D().
- Use different colors for different cities.

#### 3. Add Annotations to Highlight Extreme Weather Conditions

- Use annotate() to highlight the hottest and coldest cities.
- Label the cities with extreme humidity levels.

#### 4. Customize the 3D Scatter Plot

- Change marker size based on temperature values.
- Adjust color gradients to indicate temperature variations.

#### 5. Create a 3D Line Plot for Temperature Trends Over a Week

- Use plot3D() to visualize the temperature change over 7 days for multiple cities.
- Different lines should represent different cities.

## 6. Create a 3D Bar Chart for Average Monthly Temperature

- Use bar3D() to compare the average monthly temperature across different cities.
- Use different colors for different months.

## 7. Customize the 3D Bar Chart

- Add axis labels and titles to make the chart more readable.
- Rotate the view angle for better visualization.

## 8. Generate a Heatmap for Temperature Variation Across Months

- Use imshow() to create a heatmap where the color intensity represents temperature variations.
- Add a color bar to indicate temperature values.

## 9. Customize the Heatmap (Labels & Colors)

- Add city names to the X-axis and months to the Y-axis.
- Experiment with different color maps (coolwarm, plasma, etc.).

## 10. Compare Humidity Levels Using a Heatmap

- Create another heatmap to visualize humidity levels over time for different cities.
- Ensure it aligns with the temperature heatmap for easy comparison.

## 11. Add a Subplot to Compare Temperature & Humidity Heatmaps

- Use plt.subplots() to arrange two heatmaps (temperature & humidity) side by side.
- Adjust the layout using tight\_layout().

## 12. Create an Interactive 3D Plot for Wind Speed Analysis

- Allow users to rotate and zoom in on a 3D wind speed visualization.
- Use different markers to indicate high and low wind speeds.

### 13. Finalize the Project with a Dashboard View

- Combine all 3D plots, scatter plots, bar charts, and heatmaps into a single visualization dashboard using subplots.
- Ensure a well-structured layout and proper labeling.

## **Real-Life Mini Project 1: 3D Sales Data Visualization for an E-Commerce Company**

### **Project Overview:**

Visualize e-commerce sales data using 3D scatter plots, bar charts, and heatmaps to analyze revenue, number of orders, and customer locations.

### **Key Features:**

- Create 3D scatter plots to analyze sales trends by category, revenue, and customer count.
- Use bar3D() to compare monthly sales across multiple regions.
- Customize plots using annotations to highlight best-selling products.
- Generate a heatmap to visualize peak sales hours.

## **Real-Life Mini Project 2: 3D Air Pollution Data Analysis & Heatmap Visualization**

### **Project Overview:**

Analyze air pollution levels across multiple cities using 3D plots, scatter charts, and heatmaps to track variations in PM2.5, PM10, and NO2 levels.

### **Key Features:**

- Use scatter3D() to visualize air pollution levels (PM2.5, PM10, NO2) in different cities.
- Create a 3D bar chart to compare average monthly pollution levels.
- Add annotations to highlight highly polluted regions.
- Generate a heatmap to track air quality variations over time.

# Day 60

## Matplotlib with Pandas & Seaborn - Detailed Explanation with Examples

### 1. Introduction

Matplotlib, Pandas, and Seaborn are powerful libraries used for data visualization in Python. They help analyze, interpret, and represent data visually in different forms like line charts, bar charts, heatmaps, and pairplots.

- Matplotlib provides basic plotting functions for visualizing data.
- Pandas allows plotting directly from DataFrames using `df.plot()`.
- Seaborn enhances Matplotlib with advanced statistical visualizations like pair plots and heatmaps.

### 2. Installing Required Libraries

Before using these libraries, install them using:

```
pip install matplotlib pandas seaborn
```

### 3. Plotting Pandas DataFrames with Matplotlib

#### Definition:

Matplotlib can plot directly from Pandas DataFrames using the `.plot()` method. It is a convenient way to visualize tabular data.

#### Syntax:

```
df.plot(kind='chart_type', x='column_name', y='column_name', title='Chart Title')
```

- kind → Type of chart (line, bar, scatter, etc.)
- x → Column for X-axis
- y → Column for Y-axis
- title → Chart title

### Example: Visualizing Monthly Sales

#### Step 1: Import Libraries & Create Data

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

```
# Sample sales data
data = {
    'Month': ['Jan', 'Feb', 'Mar', 'Apr', 'May'],
    'Sales': [5000, 7000, 8000, 6500, 9000]
}
df = pd.DataFrame(data)
```

#### Step 2: Plot the Data

```
df.plot(kind='bar', x='Month', y='Sales', title='Monthly Sales', color='skyblue')
# Display the plot
plt.show()
```

**Output:** A bar chart displaying sales trends for each month.

## 4. Integrating Matplotlib with Seaborn

#### Definition:

Seaborn is built on Matplotlib and allows us to create more attractive and informative visualizations with minimal code.

#### Syntax:

```
import seaborn as sns
sns.set_style('style_name') # Sets the theme for plots
sns.barplot(x='column_name', y='column_name', data=df)



- set_style() → Changes background style (e.g., "darkgrid", "whitegrid").
- barplot() → Creates a bar chart.

```

**Example: Visualizing Sales Trends with Seaborn****Step 1: Import Seaborn & Customize Theme**

```
import seaborn as sns
```

```
# Set theme
sns.set_style("darkgrid")
```

**Step 2: Create Seaborn Bar Plot**

```
sns.barplot(x='Month', y='Sales', data=df, palette='coolwarm')
```

```
plt.title('Monthly Sales Analysis')
plt.xlabel('Month')
plt.ylabel('Total Sales')
```

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

**Output:** A better-looking bar chart with grid styling.

**5. Creating Advanced Visualizations (Pairplots & Heatmaps)****Pairplots****Definition:**

Pairplots show relationships between multiple numerical columns in a DataFrame.

**Example: Visualizing Car Data**

```
import seaborn as sns
```

```
# Load sample dataset
```

```
df_cars = sns.load_dataset('mpg')

# Create a Pairplot
sns.pairplot(df_cars[['mpg', 'horsepower', 'weight']])

plt.show()
```

**Output:** A scatterplot matrix comparing MPG, horsepower, and weight.

## Heatmaps

### Definition:

Heatmaps display correlation between numerical features in a dataset using color intensity.

### Example: Creating a Heatmap for Car Features

```
import seaborn as sns

# Compute correlation matrix
corr = df_cars[['mpg', 'horsepower', 'weight']].corr()

# Create a Heatmap
sns.heatmap(corr, annot=True, cmap='coolwarm', linewidths=0.5)

plt.title('Feature Correlation Heatmap')
plt.show()
```

**Output:** A heatmap with correlation values.

## 6. Saving & Exporting Figures

### Definition:

Matplotlib allows saving figures in different formats (PNG, JPG, PDF, etc.).

### Syntax:

```
plt.savefig('filename.format', dpi=300)
```

- dpi → Resolution quality

### Example: Save a Sales Bar Chart

```
plt.figure(figsize=(6, 4))
sns.barplot(x='Month', y='Sales', data=df, palette='viridis')
plt.title('Sales Chart')
```

```
# Save as PNG
plt.savefig('sales_chart.png', dpi=300)
```

```
plt.show()
```

**Output:** A saved image file of the sales chart.

## 7. Summary

Feature	Matplotlib	Seaborn
Basic Line & Bar Charts	✓	✓
Advanced Styling	✗	✓
Pairplots	✗	✓
Heatmaps	✗	✓

## 8. Key Takeaways

- Matplotlib provides core plotting functionalities.
- Pandas allows quick visualization from DataFrames.
- Seaborn simplifies and enhances visualizations.
- Saving figures enables report creation.

## Mini Project 1: Sales Performance Analysis Dashboard

### Objective:

Create a Sales Dashboard to visualize monthly sales, regional performance, and category-wise sales trends using Matplotlib and Seaborn.

### Step 1: Install Required Libraries

If not installed, install them using:

```
pip install pandas matplotlib seaborn
```

### Step 2: Import Libraries

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

### Step 3: Create Sample Sales Data

```
# Creating a sales dataset
data = {
    'Month': ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun'],
    'Sales': [5000, 7000, 8000, 6500, 9000, 12000],
    'Region': ['North', 'South', 'East', 'West', 'North', 'South'],
```

```

'Category': ['Electronics', 'Clothing', 'Electronics', 'Clothing', 'Electronics',
'Clothing']
}
df = pd.DataFrame(data)

```

#### **Step 4: Plot Monthly Sales using Matplotlib**

```

plt.figure(figsize=(8, 5))
plt.plot(df['Month'], df['Sales'], marker='o', linestyle='-', color='b', label='Sales')

plt.xlabel('Month')
plt.ylabel('Sales Amount')
plt.title('Monthly Sales Trend')
plt.legend()
plt.grid(True)

# Show the plot
plt.show()

```

**Output:** A line chart showing the sales trend.

#### **Step 5: Create a Bar Plot for Category-Wise Sales using Seaborn**

```

plt.figure(figsize=(8, 5))
sns.barplot(x='Category', y='Sales', data=df, palette='coolwarm')

plt.xlabel('Product Category')
plt.ylabel('Total Sales')
plt.title('Sales by Product Category')

plt.show()

```

**Output:** A bar chart showing category-wise sales.

#### Step 6: Create a Heatmap for Sales Correlation

```
# Creating a correlation matrix
df_encoded = df.copy()
df_encoded['Region'] = df_encoded['Region'].astype('category').cat.codes
df_encoded['Category'] = df_encoded['Category'].astype('category').cat.codes

corr_matrix = df_encoded.corr()

# Plot the heatmap
plt.figure(figsize=(6, 4))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', linewidths=0.5)

plt.title('Sales Data Correlation Heatmap')
plt.show()
```

**Output:** A heatmap showing correlations between sales, region, and category.

#### Step 7: Save the Figures

```
plt.figure(figsize=(8, 5))
sns.barplot(x='Category', y='Sales', data=df, palette='viridis')
plt.title('Sales by Product Category')

# Save the plot
plt.savefig('sales_by_category.png', dpi=300)
plt.show()
```

**Output:** A saved PNG file of the sales analysis chart.

## Mini Project 2: Customer Spending Behavior Analysis

### Objective:

Analyze customer spending behavior using pair plots, scatter plots, and exporting results as an image.

### Step 1: Import Libraries

(Same as before)

### Step 2: Create Sample Customer Data

```
customer_data = {
    'Customer_ID': range(1, 11),
    'Age': [22, 35, 46, 50, 29, 33, 42, 36, 28, 40],
    'Annual_Income': [30000, 60000, 80000, 70000, 50000, 75000, 90000, 72000,
    55000, 65000],
    'Spending_Score': [70, 55, 80, 50, 65, 90, 40, 85, 75, 60]
}
```

```
df_customers = pd.DataFrame(customer_data)
```

### Step 3: Create a Scatter Plot of Income vs Spending Score

```
plt.figure(figsize=(8, 5))
```

```
sns.scatterplot(x='Annual_Income', y='Spending_Score', data=df_customers,
color='red')
```

```
plt.xlabel('Annual Income')
plt.ylabel('Spending Score')
plt.title('Income vs Spending Score')
plt.show()
```

**Output:** A scatter plot showing customer income vs spending behavior.

#### **Step 4: Create a Pair Plot for Customer Attributes**

```
sns.pairplot(df_customers, diag_kind='kde')  
  
plt.show()
```

**Output:** A pairplot matrix showing relationships between Age, Income, and Spending Score.

#### **Step 5: Save the Scatter Plot**

```
plt.figure(figsize=(8, 5))  
sns.scatterplot(x='Annual_Income', y='Spending_Score', data=df_customers,  
color='blue')  
plt.title('Income vs Spending Score')  
  
# Save the figure  
plt.savefig('income_vs_spending.png', dpi=300)  
plt.show()
```

**Output:** A saved PNG file of the income vs spending scatter plot.

## Mini Project: Employee Performance & Salary Analysis

### Objective:

Analyze employee performance and salary trends using Matplotlib, Pandas, and Seaborn to gain insights into salary distributions, experience vs salary relationships, and department-wise salary trends.

### Tasks:

1. **Load Employee Dataset:** Import a dataset containing employee salaries, experience, department, and performance scores using Pandas.
2. **Data Preprocessing:** Check for missing values and handle them appropriately.
3. **Plot Salary Distribution:** Use a histogram (`df.plot(kind='hist')`) to visualize the distribution of employee salaries.
4. **Department-wise Salary Bar Chart:** Create a **bar chart** to display the average salary per department using Matplotlib.
5. **Experience vs Salary Scatter Plot:** Use Seaborn's `sns.scatterplot()` to visualize the relationship between experience and salary.
6. **Box Plot for Salary Comparison:** Generate a **box plot** to compare salary distributions across different departments.
7. **Pairplot for Performance Analysis:** Use `sns.pairplot()` to explore relationships between salary, experience, and performance scores.
8. **Heatmap for Correlation Analysis:** Generate a **heatmap** to identify correlations between numerical columns (Salary, Experience, Performance Score).
9. **Save the Salary Distribution Chart:** Export the **salary histogram** as a **PNG file**.

10. **Customize the Scatter Plot:** Modify **point sizes and colors** in the experience vs salary scatter plot.
11. **Adjust Subplot Layouts:** Create a **subplot** layout with different visualizations in a single figure.
12. **Export the Final Dashboard:** Save all figures in **PDF and PNG formats** for reporting.
13. **Generate Summary Report:** Display key insights from the visualizations and suggest trends in salary distribution and performance.

This project **blends Matplotlib with Seaborn** for professional-grade **data visualization and analysis**. ↗

### Mini Project 1: Website Traffic Analysis & Visualization

Objective:

Analyze **website traffic data** using Matplotlib, Pandas, and Seaborn to understand user engagement, peak hours, and visitor trends.

Requirements:

1. Load a dataset containing **daily website visits, unique visitors, page views, bounce rate, and average session duration**.
2. Clean and preprocess the data by handling missing values and formatting dates.
3. Create a **line plot** using `df.plot()` to visualize daily website visits over time.
4. Generate a **bar chart** to show the number of visits per weekday.
5. Use a **scatter plot** to explore the relationship between page views and average session duration.
6. Create a **box plot** to compare bounce rates across different weekdays.

7. Generate a **pairplot** using Seaborn to analyze correlations between different engagement metrics.
8. Build a **heatmap** to visualize correlations between visits, bounce rate, and session duration.
9. Add titles, labels, and annotations to improve plot readability.
10. Save and export each visualization in **PNG format** for analytics reports.

## Mini Project 2: Stock Market Data Visualization

Objective:

Analyze **stock market data** using Matplotlib, Pandas, and Seaborn to track trends, volatility, and stock performance.

Requirements:

1. Load a dataset containing **date, stock prices (open, close, high, low), trading volume, and percentage change**.
2. Clean and preprocess the data, ensuring date formatting and handling missing values.
3. Create a **line plot** to visualize daily stock price fluctuations.
4. Generate a **bar chart** to show daily trading volumes.
5. Use a **scatter plot** to analyze the relationship between trading volume and percentage price change.
6. Create a **box plot** to compare stock price distributions over different months.
7. Generate a **pairplot** using Seaborn to analyze relationships between different stock indicators.
8. Create a **heatmap** to examine correlations between stock price changes, trading volume, and market trends.
9. Add necessary labels, legends, and titles to enhance clarity.
10. Export all visualizations in **JPEG format** for investment analysis reports.

# Seaborn

## Day 61

### Introduction to Seaborn & Basic Plots

#### What is Seaborn? Why use it for Data Visualization?

##### Definition:

Seaborn is a **Python data visualization library** built on top of Matplotlib. It provides an interface for creating **attractive, informative, and statistical graphics**.

#### Why Use Seaborn?

1. **High-Level Interface** – It simplifies the process of plotting complex visualizations.
2. **Built-in Themes** – Provides aesthetic and visually appealing styles.
3. **Integrated with Pandas** – Works directly with Pandas DataFrames.
4. **Statistical Visualizations** – Supports advanced plots like regression plots, violin plots, and heatmaps.
5. **Better Color Handling** – Automatically assigns colors based on categories.

#### Installing Seaborn

To install Seaborn, use the following command:

```
pip install seaborn
```

## Importing Seaborn

Before using Seaborn, import it as follows:

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

## Loading Built-in Datasets

Seaborn provides built-in datasets that can be used for practice.

### 1. View Available Datasets

```
import seaborn as sns  
  
print(sns.get_dataset_names())
```

This will return a list of built-in datasets like tips, iris, penguins, diamonds, etc.

### 2. Load a Dataset

```
df = sns.load_dataset("tips") # Loads the "tips" dataset  
print(df.head()) # Display the first five rows
```

## Basic Plots in Seaborn

Seaborn provides functions for different types of plots. Let's explore:

## Line Plot (sns.lineplot())

A **line plot** is used to show trends over time.

### Example: Sales Trend Analysis

Step-by-Step Implementation:

- 1. Import required libraries**
- 2. Create a dataset or load a built-in dataset**
- 3. Use sns.lineplot() to create a line chart**
- 4. Customize title, labels, and legend**
- 5. Display the plot**

```
import seaborn as sns
```

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

```
import pandas as pd
```

```
# Creating a dataset
```

```
data = pd.DataFrame({  
    "month": ["Jan", "Feb", "Mar", "Apr", "May", "Jun"],  
    "sales": [100, 150, 180, 220, 300, 400]  
})
```

```
# Creating a line plot
```

```
sns.lineplot(x="month", y="sales", data=data, marker="o", color="blue",  
            linewidth=2)
```

```
# Customizing the plot
```

```
plt.title("Monthly Sales Trend", fontsize=14)  
plt.xlabel("Month", fontsize=12)  
plt.ylabel("Sales ($)", fontsize=12)
```

```
plt.grid(True)
```

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

### Explanation:

- `sns.lineplot()` → Plots the sales trend over months.
- `marker="o"` → Adds circular markers to each point.
- `color="blue"` → Sets the line color.
- `linewidth=2` → Adjusts line thickness.
- `plt.grid(True)` → Adds a background grid for better readability.

### Scatter Plot (`sns.scatterplot()`)

A **scatter plot** is used to visualize the relationship between two numerical variables.

### Example: Relationship between Bill and Tip Amount

Step-by-Step Implementation:

```
import seaborn as sns  
import matplotlib.pyplot as plt  
  
# Load dataset  
df = sns.load_dataset("tips")  
  
# Create scatter plot  
sns.scatterplot(x="total_bill", y="tip", data=df, hue="sex", style="sex", s=100)
```

```
# Customizing the plot  
plt.title("Total Bill vs Tip Amount", fontsize=14)  
plt.xlabel("Total Bill ($)", fontsize=12)  
plt.ylabel("Tip Amount ($)", fontsize=12)  
plt.legend(title="Gender")  
  
# Show plot  
plt.show()
```

### Explanation:

- `sns.scatterplot()` → Creates a scatter plot.
- `x="total_bill", y="tip"` → Plots the relationship between total bill and tip amount.
- `hue="sex"` → Colors points based on gender (Male/Female).
- `style="sex"` → Uses different markers for Male and Female.
- `s=100` → Adjusts marker size.

### Bar Plot (`sns.barplot()`)

A **bar plot** is used to compare categorical data.

### Example: Average Tip Amount by Day

Step-by-Step Implementation:

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

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

# Create bar plot
sns.barplot(x="day", y="tip", data=df, palette="Blues")

# Customizing the plot
plt.title("Average Tip Amount by Day", fontsize=14)
plt.xlabel("Day of the Week", fontsize=12)
plt.ylabel("Average Tip ($)", fontsize=12)

# Show plot
plt.show()
```

### Explanation:

- ✓ **sns.barplot()** → Creates a bar plot.
- ✓ **x="day", y="tip"** → Plots average tip amount per day.
- ✓ **palette="Blues"** → Sets color theme.

### ◆ Customizing Titles, Labels, and Legends

To make plots **more readable**, we can customize:

- **Titles:** plt.title("Title", fontsize=14)
- **X-axis label:** plt.xlabel("X Label", fontsize=12)
- **Y-axis label:** plt.ylabel("Y Label", fontsize=12)

- **Legend:** plt.legend(title="Legend Title")
- **Grid:** plt.grid(True)

Example of adding these customizations:

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

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

# Create scatter plot
sns.scatterplot(x="total_bill", y="tip", data=df, hue="sex", style="sex", s=100)

# Customizing titles, labels, and legend
plt.title("Total Bill vs Tip Amount", fontsize=14)
plt.xlabel("Total Bill ($)", fontsize=12)
plt.ylabel("Tip Amount ($)", fontsize=12)
plt.legend(title="Gender")

# Show plot
plt.show()
```

## Summary

Feature	Function	Purpose
<b>Installing Seaborn</b>	pip install seaborn	Installs Seaborn library
<b>Importing Seaborn</b>	import seaborn as sns	Imports Seaborn into Python

<b>Loading Dataset</b>	df = sns.load_dataset("dataset_name")	Loads a built-in dataset
<b>Line Plot</b>	sns.lineplot()	Shows trends over time
<b>Scatter Plot</b>	sns.scatterplot()	Shows relationships between two numerical variables
<b>Bar Plot</b>	sns.barplot()	Compares categorical data
<b>Customizing Titles &amp; Labels</b>	plt.title(), plt.xlabel(), plt.ylabel()	Enhances plot readability
<b>Adding Grid</b>	plt.grid(True)	Improves visual clarity

## Real-World Applications

1. **Sales & Revenue Analysis** – Track trends using sns.lineplot().
2. **Customer Spending Patterns** – Understand tip behavior using sns.scatterplot().
3. **Market Research** – Compare sales performance across categories using sns.barplot().

## Mini Project 1: Sales Performance Analysis Using Seaborn

### Problem Statement:

A retail company wants to analyze its **monthly sales performance** for two different product categories. The goal is to visualize the **sales trends, compare category-wise performance**, and identify **insights for future decisions** using Seaborn.

## Steps to Implement:

### Step 1: Install and Import Required Libraries

```
# Install Seaborn if not installed  
!pip install seaborn
```

```
# Import necessary libraries  
import seaborn as sns  
import matplotlib.pyplot as plt  
import pandas as pd
```

### Step 2: Create a Sample Dataset

```
# Create a sample dataset  
data = pd.DataFrame({  
    "Month": ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct",  
    "Nov", "Dec"],  
    "Category": ["Electronics", "Electronics", "Electronics", "Electronics",  
    "Electronics", "Electronics",  
    "Clothing", "Clothing", "Clothing", "Clothing", "Clothing", "Clothing"],  
    "Sales": [12000, 15000, 18000, 22000, 25000, 28000, 8000, 10000, 12000,  
    15000, 17000, 20000]  
})  
  
# Display the dataset  
print(data.head())
```

### Step 3: Line Plot - Monthly Sales Trend

```
# Create a line plot for sales trend
plt.figure(figsize=(10, 5))
sns.lineplot(x="Month", y="Sales", hue="Category", data=data, marker="o",
linewidth=2)

# Customizing the plot
plt.title("Monthly Sales Trend", fontsize=14)
plt.xlabel("Month", fontsize=12)
plt.ylabel("Sales ($)", fontsize=12)
plt.xticks(rotation=45)
plt.grid(True)

# Show plot
plt.show()
```

#### Explanation:

- ✓ sns.lineplot() → Creates a line plot with months on X-axis and sales on Y-axis.
- ✓ hue="Category" → Differentiates Electronics & Clothing using colors.
- ✓ marker="o" → Adds circle markers for better visibility.
- ✓ plt.xticks(rotation=45) → Rotates month labels for readability.

## Step 4: Scatter Plot - Relationship Between Category & Sales

```
# Create a scatter plot
plt.figure(figsize=(8, 5))
sns.scatterplot(x="Month", y="Sales", hue="Category", style="Category",
                 data=data, s=100)

# Customizing the plot
plt.title("Category-wise Sales Distribution", fontsize=14)
plt.xlabel("Month", fontsize=12)
plt.ylabel("Sales ($)", fontsize=12)
plt.xticks(rotation=45)
plt.legend(title="Product Category")

# Show plot
plt.show()
```

### Explanation:

- ✓ sns.scatterplot() → Displays how sales vary per month for each category.
- ✓ style="Category" → Uses different markers for categories.
- ✓ s=100 → Increases marker size for better visualization.

## Step 5: Bar Plot - Comparing Average Sales per Category

```
# Create a bar plot for average sales
plt.figure(figsize=(6, 4))
sns.barplot(x="Category", y="Sales", data=data, palette="coolwarm")
```

```
# Customizing the plot  
plt.title("Average Sales by Category", fontsize=14)  
plt.xlabel("Product Category", fontsize=12)  
plt.ylabel("Average Sales ($)", fontsize=12)  
  
# Show plot  
plt.show()
```

### Explanation:

- ✓ sns.barplot() → Shows the average sales for each category.
- ✓ palette="coolwarm" → Adds a color theme for better appearance.

### Insights from the Plots:

1. **Electronics sales** are **higher** and show an **upward trend** over the months.
2. **Clothing sales** are **lower** but increase steadily over time.
3. The **scatter plot confirms** that electronics have **higher monthly sales variability**.
4. The **bar plot shows** that Electronics have a **higher average sales** compared to Clothing.

## Mini Project 2: Restaurant Tips Analysis Using Seaborn

### Problem Statement:

A restaurant wants to analyze how **total bill amounts influence tips**. The management also wants to know whether **gender** affects tipping behavior.

### Steps to Implement:

#### Step 1: Install and Import Required Libraries

```
# Install Seaborn if not installed  
!pip install seaborn
```

```
# Import necessary libraries  
import seaborn as sns  
import matplotlib.pyplot as plt
```

#### Step 2: Load the Built-in "tips" Dataset

```
# Load the built-in "tips" dataset  
df = sns.load_dataset("tips")  
  
# Display first 5 rows  
print(df.head())
```

#### Step 3: Scatter Plot - Relationship Between Total Bill and Tips

```
# Create a scatter plot  
plt.figure(figsize=(8, 5))  
sns.scatterplot(x="total_bill", y="tip", hue="sex", style="sex", data=df, s=100)
```

```
# Customizing the plot
plt.title("Total Bill vs Tip Amount", fontsize=14)
plt.xlabel("Total Bill ($)", fontsize=12)
plt.ylabel("Tip Amount ($)", fontsize=12)
plt.legend(title="Gender")

# Show plot
plt.show()
```

**Explanation:**

- ✓ Shows how total bill influences tip amount.
- ✓ Different colors & styles for Male and Female customers.
- ✓ Marker size increased for better visibility.

**Step 4: Line Plot - Trends in Tip Amount Over Time**

```
# Create a line plot
plt.figure(figsize=(8, 5))
sns.lineplot(x="total_bill", y="tip", data=df, color="green", marker="o")

# Customizing the plot
plt.title("Tip Amount Trends with Total Bill", fontsize=14)
plt.xlabel("Total Bill ($)", fontsize=12)
plt.ylabel("Tip Amount ($)", fontsize=12)
plt.grid(True)

# Show plot
plt.show()
```

**Explanation:**

- ✓ Shows tip trends based on total bill amount.
- ✓ Helps identify if larger bills lead to higher tips.
- ✓ Grid added for better readability.

**Step 5: Bar Plot - Average Tip by Gender**

```
# Create a bar plot
plt.figure(figsize=(6, 4))
sns.barplot(x="sex", y="tip", data=df, palette="pastel")

# Customizing the plot
plt.title("Average Tip Amount by Gender", fontsize=14)
plt.xlabel("Gender", fontsize=12)
plt.ylabel("Average Tip ($)", fontsize=12)

# Show plot
plt.show()
```

**Explanation:**

- ✓ Compares average tip amounts between Male and Female customers.
- ✓ Uses pastel colors for better aesthetics.

## Insights from the Plots:

1. Higher total bills generally result in higher tips.
2. Male and Female customers show slight differences in tipping behavior.
3. The bar plot indicates whether gender impacts the average tip amount.
4. Management can use these insights to optimize pricing & customer engagement strategies.

## Summary of Concepts Used in Both Projects

Feature	Function	Purpose
<b>Line Plot</b>	sns.lineplot()	Shows trends over time
<b>Scatter Plot</b>	sns.scatterplot()	Displays relationships between two variables
<b>Bar Plot</b>	sns.barplot()	Compares categorical data
<b>Dataset Loading</b>	sns.load_dataset()	Loads built-in datasets
<b>Customizing Plots</b>	plt.title(), plt.xlabel(), plt.ylabel()	Enhances readability
<b>Legends &amp; Labels</b>	plt.legend(), plt.grid(True)	Adds clarity to plots

## Real-Life Mini Project: Employee Performance & Salary Analysis using Seaborn

Problem Statement:

A company wants to analyze employee performance and salary trends based on experience level. The goal is to **visualize salary trends, compare gender-based salary differences, and analyze performance over time** using Seaborn.

### Project Tasks

#### 1. What is Seaborn? Why use it for Data Visualization?

- Explain the advantages of using Seaborn over Matplotlib.

#### 2. Installing and Importing Seaborn

- Install Seaborn (pip install seaborn) and import necessary libraries.

#### 3. Loading & Exploring the Employee Dataset

- Load an employee dataset containing **experience level, salary, performance rating, and gender**.

#### 4. Displaying Summary Statistics of the Dataset

- Use .info() and .describe() to understand data structure.

#### 5. Line Plot - Salary Growth Over Time

- Create a line plot (sns.lineplot()) to show how salaries increase with experience.

## 6. Scatter Plot - Relationship Between Performance & Salary

- Create a scatter plot (`sns.scatterplot()`) to visualize the correlation between salary and performance rating.

## 7. Bar Plot - Average Salary Based on Experience Level

- Use a bar plot (`sns.barplot()`) to compare average salaries at different experience levels.

## 8. Gender-Based Salary Comparison

- Use `sns.barplot()` to analyze if there's a salary gap between male and female employees.

## 9. Customizing Titles, Labels, and Legends

- Add titles, axis labels, and legends to make visualizations clear.

## 10. Changing Color Palettes for Better Visualization

- Experiment with different Seaborn color palettes (`coolwarm`, `viridis`, `pastel`).

## 11. Adding Grid and Style to Plots

- Apply `sns.set_style()` to enhance the aesthetic appearance of the plots.

## 12. Exporting the Visualizations

- Save the plots as PNG or PDF using `plt.savefig()`.

## 13. Insights & Business Recommendations

- Interpret the visualizations and provide recommendations for salary structuring and performance evaluation.

## **Mini Project 1: Sales Trend Analysis for an E-commerce Business**

Project Requirement:

An e-commerce company wants to analyze monthly sales trends and customer spending behavior using Seaborn. The project should visualize monthly revenue trends, scatter plots for customer purchases, and bar charts for product category sales to gain insights into business growth.

## **Mini Project 2: Student Performance Analysis in an Educational Institution**

Project Requirement:

A school wants to analyze students' academic performance using Seaborn. The project should include line plots for average student scores over time, scatter plots for attendance vs. performance, and bar charts for subject-wise performance to identify learning patterns and areas of improvement.

# Day 62

## Categorical Data Visualization in Seaborn

Categorical data visualization helps us understand the distribution, frequency, and relationships of categorical variables in a dataset. Seaborn provides multiple plot types for this purpose.

### 1. Bar Plot (sns.barplot())

#### Definition:

A bar plot represents categorical data with bars whose heights are based on the aggregated statistical measure (default is the mean).

#### Syntax:

```
sns.barplot(x='category_column', y='numerical_column', data=df)
```

#### Real-life Example:

**Scenario:** A company wants to analyze the **average salary of employees** in different departments.

#### Step-by-Step Implementation:

```
import seaborn as sns  
import matplotlib.pyplot as plt  
import pandas as pd  
  
# Sample dataset
```

```
data = {'Department': ['HR', 'IT', 'Finance', 'Marketing', 'IT', 'HR', 'Finance',  
'Marketing'],  
       'Salary': [50000, 70000, 60000, 55000, 75000, 52000, 62000, 58000]}  
  
df = pd.DataFrame(data)  
  
# Creating a bar plot  
sns.barplot(x='Department', y='Salary', data=df, palette='Blues')  
  
# Customizing the plot  
plt.title('Average Salary by Department')  
plt.xlabel('Department')  
plt.ylabel('Salary')  
plt.show()
```

**Insights:** This will display **average salaries** for each department, helping HR understand salary distribution.

## 2. Count Plot (sns.countplot())

### Definition:

A count plot shows the frequency of categorical variables.

### Syntax:

```
sns.countplot(x='category_column', data=df)
```

## Real-life Example:

**Scenario:** A restaurant wants to analyze **how many orders come from each city**.

## Step-by-Step Implementation:

```
data = {'City': ['New York', 'Los Angeles', 'Chicago', 'New York', 'Chicago', 'Los  
Angeles', 'New York', 'Chicago']}
```

```
df = pd.DataFrame(data)
```

```
sns.countplot(x='City', data=df, palette='coolwarm')
```

```
plt.title('Number of Orders by City')  
plt.xlabel('City')  
plt.ylabel('Count')  
plt.show()
```

**Insights:** Helps restaurants identify which cities have the most orders.

## 3. Box Plot (sns.boxplot())

### Definition:

A box plot shows the **distribution of numerical data** and identifies **outliers**.

### Syntax:

```
sns.boxplot(x='category_column', y='numerical_column', data=df)
```

## Real-life Example:

**Scenario:** A school wants to analyze **students' test scores** in different subjects.

## Step-by-Step Implementation:

```
data = {'Subject': ['Math', 'Science', 'English', 'Math', 'Science', 'English', 'Math',  
'Science'],  
       'Score': [85, 78, 92, 88, 74, 89, 95, 80]}  
  
df = pd.DataFrame(data)  
  
sns.boxplot(x='Subject', y='Score', data=df, palette='Set2')  
  
plt.title('Student Test Scores Distribution')  
plt.xlabel('Subject')  
plt.ylabel('Score')  
plt.show()
```

**Insights:** Identifies **outliers** (students scoring very high or low).

## 4. Violin Plot (sns.violinplot())

### Definition:

A violin plot is like a box plot but also shows **the density of data distribution**.

## Syntax:

```
sns.violinplot(x='category_column', y='numerical_column', data=df)
```

## Real-life Example:

**Scenario:** A fitness club wants to analyze **members' weight distribution** in different age groups.

## Step-by-Step Implementation:

```
data = {'Age Group': ['20-30', '30-40', '40-50', '20-30', '30-40', '40-50', '20-30', '30-40'],
        'Weight': [65, 78, 82, 70, 80, 85, 68, 77]}
```

```
df = pd.DataFrame(data)
```

```
sns.violinplot(x='Age Group', y='Weight', data=df, palette='muted')
```

```
plt.title('Weight Distribution by Age Group')
plt.xlabel('Age Group')
plt.ylabel('Weight')
plt.show()
```

**Insights:** Shows where most data points are concentrated.

## 5. Swarm Plot (sns.swarmplot())

### Definition:

A swarm plot shows **individual data points** without overlapping.

### Syntax:

```
sns.swarmplot(x='category_column', y='numerical_column', data=df)
```

### Real-life Example:

**Scenario:** A company wants to analyze **employee experience levels** in different job roles.

### Step-by-Step Implementation:

```
data = {'Job Role': ['Manager', 'Developer', 'Analyst', 'Manager', 'Developer',
'Analyst', 'Manager', 'Developer'],
'Experience': [10, 3, 5, 12, 4, 6, 11, 2]}
```

```
df = pd.DataFrame(data)
```

```
sns.swarmplot(x='Job Role', y='Experience', data=df, palette='coolwarm')
```

```
plt.title('Employee Experience by Job Role')
plt.xlabel('Job Role')
plt.ylabel('Years of Experience')
plt.show()
```

**Insights:** Helps in understanding **employee experience distribution**.

## 6. Customizing Colors & Styles

Seaborn allows you to **customize colors** to improve readability.

### Changing Color Palette

```
sns.set_palette('pastel')
```

### Customizing Background Style

```
sns.set_style('darkgrid')
```

### Applying Custom Themes

```
sns.set_theme(style="whitegrid", palette="pastel")
```

### Summary

Plot Type	Use Case
<b>Bar Plot</b> (sns.barplot())	Compare numerical values across categories
<b>Count Plot</b> (sns.countplot())	Show frequency distribution of categorical values
<b>Box Plot</b> (sns.boxplot())	Identify data distribution and outliers
<b>Violin Plot</b> (sns.violinplot())	Show distribution and density
<b>Swarm Plot</b> (sns.swarmplot())	Visualize individual data points without overlap

## Categorical Data Visualization in Seaborn

Categorical data visualization helps us understand the distribution, frequency, and relationships of categorical variables in a dataset. Seaborn provides multiple plot types for this purpose.

### 1. Bar Plot (sns.barplot())

#### Definition:

A bar plot represents categorical data with bars whose heights are based on the aggregated statistical measure (default is the mean).

#### Syntax:

```
sns.barplot(x='category_column', y='numerical_column', data=df)
```

#### Real-life Example:

**Scenario:** A company wants to analyze the **average salary of employees** in different departments.

#### Step-by-Step Implementation:

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

# Sample dataset
data = {'Department': ['HR', 'IT', 'Finance', 'Marketing', 'IT', 'HR', 'Finance',
```

```
'Marketing'],
'Salary': [50000, 70000, 60000, 55000, 75000, 52000, 62000, 58000]}

df = pd.DataFrame(data)

# Creating a bar plot
sns.barplot(x='Department', y='Salary', data=df, palette='Blues')

# Customizing the plot
plt.title('Average Salary by Department')
plt.xlabel('Department')
plt.ylabel('Salary')
plt.show()
```

**Insights:** This will display **average salaries** for each department, helping HR understand salary distribution.

## 2. Count Plot (sns.countplot())

### Definition:

A count plot shows the frequency of categorical variables.

### Syntax:

```
sns.countplot(x='category_column', data=df)
```

## Real-life Example:

**Scenario:** A restaurant wants to analyze **how many orders come from each city**.

## Step-by-Step Implementation:

```
data = {'City': ['New York', 'Los Angeles', 'Chicago', 'New York', 'Chicago', 'Los  
Angeles', 'New York', 'Chicago']}
```

```
df = pd.DataFrame(data)
```

```
sns.countplot(x='City', data=df, palette='coolwarm')
```

```
plt.title('Number of Orders by City')  
plt.xlabel('City')  
plt.ylabel('Count')  
plt.show()
```

**Insights:** Helps restaurants identify which cities have the most orders.

## 3. Box Plot (sns.boxplot())

### Definition:

A box plot shows the **distribution of numerical data** and identifies **outliers**.

### Syntax:

```
sns.boxplot(x='category_column', y='numerical_column', data=df)
```

## Real-life Example:

**Scenario:** A school wants to analyze **students' test scores** in different subjects.

## Step-by-Step Implementation:

```
data = {'Subject': ['Math', 'Science', 'English', 'Math', 'Science', 'English', 'Math',  
'Science'],  
       'Score': [85, 78, 92, 88, 74, 89, 95, 80]}  
  
df = pd.DataFrame(data)  
  
sns.boxplot(x='Subject', y='Score', data=df, palette='Set2')  
  
plt.title('Student Test Scores Distribution')  
plt.xlabel('Subject')  
plt.ylabel('Score')  
plt.show()
```

**Insights:** Identifies **outliers** (students scoring very high or low).

## 4. Violin Plot (sns.violinplot())

### Definition:

A violin plot is like a box plot but also shows **the density of data distribution**.

## Syntax:

```
sns.violinplot(x='category_column', y='numerical_column', data=df)
```

## Real-life Example:

**Scenario:** A fitness club wants to analyze **members' weight distribution** in different age groups.

## Step-by-Step Implementation:

```
data = {'Age Group': ['20-30', '30-40', '40-50', '20-30', '30-40', '40-50', '20-30', '30-40'],
        'Weight': [65, 78, 82, 70, 80, 85, 68, 77]}
```

```
df = pd.DataFrame(data)
```

```
sns.violinplot(x='Age Group', y='Weight', data=df, palette='muted')
```

```
plt.title('Weight Distribution by Age Group')
plt.xlabel('Age Group')
plt.ylabel('Weight')
plt.show()
```

**Insights:** Shows where most data points are concentrated.

## 5. Swarm Plot (sns.swarmplot())

### Definition:

A swarm plot shows **individual data points** without overlapping.

### Syntax:

```
sns.swarmplot(x='category_column', y='numerical_column', data=df)
```

### Real-life Example:

**Scenario:** A company wants to analyze **employee experience levels** in different job roles.

### Step-by-Step Implementation:

```
data = {'Job Role': ['Manager', 'Developer', 'Analyst', 'Manager', 'Developer',
'Analyst', 'Manager', 'Developer'],
'Experience': [10, 3, 5, 12, 4, 6, 11, 2]}
```

```
df = pd.DataFrame(data)
```

```
sns.swarmplot(x='Job Role', y='Experience', data=df, palette='coolwarm')
```

```
plt.title('Employee Experience by Job Role')
plt.xlabel('Job Role')
plt.ylabel('Years of Experience')
plt.show()
```

**Insights:** Helps in understanding **employee experience distribution**.

## 6. Customizing Colors & Styles

Seaborn allows you to **customize colors** to improve readability.

### Changing Color Palette

```
sns.set_palette('pastel')
```

### Customizing Background Style

```
sns.set_style('darkgrid')
```

### Applying Custom Themes

```
sns.set_theme(style="whitegrid", palette="pastel")
```

### Summary

Plot Type	Use Case
<b>Bar Plot</b> (sns.barplot())	Compare numerical values across categories
<b>Count Plot</b> (sns.countplot())	Show frequency distribution of categorical values
<b>Box Plot</b> (sns.boxplot())	Identify data distribution and outliers
<b>Violin Plot</b> (sns.violinplot())	Show distribution and density
<b>Swarm Plot</b> (sns.swarmplot())	Visualize individual data points without overlap

## Mini Project 1: Employee Salary Analysis Using Categorical Plots

### Objective:

Analyze salary distribution across different job roles using Seaborn's categorical plots.

### Step 1: Install & Import Necessary Libraries

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

### Step 2: Create a Sample Dataset

```
# Creating a dataset  
data = {  
    'Job Role': ['Manager', 'Developer', 'Designer', 'Analyst', 'Manager', 'Developer',  
    'Designer', 'Analyst'],  
    'Salary': [95000, 70000, 65000, 72000, 98000, 73000, 68000, 75000],  
    'Experience': [10, 5, 4, 6, 12, 7, 3, 8]  
}  
  
df = pd.DataFrame(data)
```

### Columns:

- **Job Role** (Categorical)
- **Salary** (Numerical)
- **Experience** (Numerical)

### Step 3: Bar Plot – Average Salary per Job Role

```
plt.figure(figsize=(8,5))
sns.barplot(x='Job Role', y='Salary', data=df, palette='coolwarm')

plt.title('Average Salary per Job Role')
plt.xlabel('Job Role')
plt.ylabel('Salary')
plt.show()
```

**Insights:** Helps compare **average salaries** across job roles.

### Step 4: Count Plot – Number of Employees in Each Role

```
plt.figure(figsize=(8,5))
sns.countplot(x='Job Role', data=df, palette='pastel')

plt.title('Number of Employees in Each Role')
plt.xlabel('Job Role')
plt.ylabel('Count')
plt.show()
```

**Insights:** Shows **how many employees** are in each job category.

### Step 5: Box Plot – Salary Distribution & Outliers

```
plt.figure(figsize=(8,5))
sns.boxplot(x='Job Role', y='Salary', data=df, palette='Set2')

plt.title('Salary Distribution per Job Role')
plt.xlabel('Job Role')
```

```
plt.ylabel('Salary')
plt.show()
```

**Insights:** Helps **detect outliers** in salary distribution.

### Step 6: Violin Plot – Salary Density per Job Role

```
plt.figure(figsize=(8,5))
sns.violinplot(x='Job Role', y='Salary', data=df, palette='muted')
```

```
plt.title('Salary Distribution & Density per Job Role')
plt.xlabel('Job Role')
plt.ylabel('Salary')
plt.show()
```

**Insights:** Shows **salary variation and density**.

### Step 7: Swarm Plot – Salary vs. Job Role

```
plt.figure(figsize=(8,5))
sns.swarmplot(x='Job Role', y='Salary', data=df, palette='coolwarm')
```

```
plt.title('Individual Salary Data Points')
plt.xlabel('Job Role')
plt.ylabel('Salary')
plt.show()
```

**Insights:** Displays **individual salary points**

## Step 8: Customizing Colors & Styles

```
sns.set_style('whitegrid')
sns.set_palette('pastel')
```

**Effect:** Enhances plot readability.

## Mini Project 2: Sales Performance Analysis Using Categorical Plots

### Objective:

Analyze sales performance across different product categories using Seaborn.

### Step 1: Install & Import Libraries

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

### Step 2: Create a Sample Dataset

```
data = {
    'Product Category': ['Electronics', 'Clothing', 'Groceries', 'Furniture',
    'Electronics', 'Clothing', 'Groceries', 'Furniture'],
    'Sales': [50000, 30000, 20000, 45000, 55000, 32000, 22000, 47000],
    'Profit': [5000, 7000, 3000, 6000, 5200, 7200, 3100, 6500]
}

df = pd.DataFrame(data)
```

### Step 3: Bar Plot – Average Sales per Category

```
plt.figure(figsize=(8,5))
sns.barplot(x='Product Category', y='Sales', data=df, palette='coolwarm')

plt.title('Average Sales per Product Category')
plt.xlabel('Product Category')
plt.ylabel('Sales')
plt.show()
```

**Insights:** Compares **sales across categories**.

### Step 4: Count Plot – Number of Transactions per Category

```
plt.figure(figsize=(8,5))
sns.countplot(x='Product Category', data=df, palette='pastel')

plt.title('Number of Transactions per Product Category')
plt.xlabel('Product Category')
plt.ylabel('Count')
plt.show()
```

**Insights:** Helps identify **popular product categories**.

### Step 5: Box Plot – Sales Distribution

```
plt.figure(figsize=(8,5))
sns.boxplot(x='Product Category', y='Sales', data=df, palette='Set2')

plt.title('Sales Distribution per Product Category')
plt.xlabel('Product Category')
```

```
plt.ylabel('Sales')
plt.show()
```

**Insights:** Identifies **sales outliers**.

## Step 6: Violin Plot – Profit Distribution

```
plt.figure(figsize=(8,5))
sns.violinplot(x='Product Category', y='Profit', data=df, palette='muted')
```

```
plt.title('Profit Distribution per Product Category')
plt.xlabel('Product Category')
plt.ylabel('Profit')
plt.show()
```

**Insights:** Shows **profit density variation**.

## Step 7: Swarm Plot – Sales vs. Product Category

```
plt.figure(figsize=(8,5))
sns.swarmplot(x='Product Category', y='Sales', data=df, palette='coolwarm')
```

```
plt.title('Sales Data Points per Product Category')
plt.xlabel('Product Category')
plt.ylabel('Sales')
plt.show()
```

**Insights:** Displays **individual sales points**.

## Step 8: Customizing Colors & Styles

```
sns.set_style('darkgrid')  
sns.set_palette('pastel')
```

**Effect:** Enhances visual aesthetics.

### Key Takeaways

- ✓ **Bar Plots** compare **averages** across categories.
- ✓ **Count Plots** show **frequency** of categorical values.
- ✓ **Box Plots** detect **outliers** in distributions.
- ✓ **Violin Plots** show **density variation**.
- ✓ **Swarm Plots** help visualize **individual data points**.

## Mini Project: Customer Purchase Behavior Analysis Using Categorical Plots

### Objective:

Analyze customer purchase patterns and spending habits using Seaborn's categorical plots.

### Tasks:

1. **Load and Explore Data:** Load a dataset containing customer purchase details (Product Category, Purchase Amount, Payment Method, etc.).
2. **Data Cleaning:** Handle missing values and remove duplicates if necessary.
3. **Bar Plot – Average Purchase Amount per Category:** Use sns.barplot() to compare spending across different product categories.
4. **Count Plot – Number of Purchases per Payment Method:** Use sns.countplot() to visualize payment preferences (e.g., Credit Card, Cash, UPI).
5. **Box Plot – Distribution of Purchase Amounts:** Use sns.boxplot() to detect outliers in spending habits.
6. **Violin Plot – Spending Distribution by Product Category:** Use sns.violinplot() to observe variations in spending.
7. **Swarm Plot – Individual Purchase Amounts per Category:** Use sns.swarmplot() to plot individual transactions.
8. **Compare Purchase Amounts by Customer Age Group:** Use sns.barplot() or sns.boxplot() to analyze spending patterns across different age groups.
9. **Analyze Seasonal Trends in Purchases:** Use a bar plot to compare purchases made during different months or seasons.
10. **Segment Customers Based on Spending Habits:** Categorize customers into Low, Medium, and High spenders and visualize the distribution.

11. **Visualize Discounts & Offers Impact on Purchase Behavior:** Use a box plot to compare spending before and after discounts.
12. **Customize Plot Styles and Themes:** Modify Seaborn color palettes and styles to improve readability.
13. **Save and Export Final Visualizations:** Save plots as images (.png or .jpg) for reports or presentations.

## Mini Project 1: Online Course Enrollment Analysis

**Objective:** Analyze student enrollment patterns and course popularity using categorical plots in Seaborn.

- ◆ Use **bar plots** to compare the average number of students enrolled in different course categories (e.g., Data Science, Web Development, Business).
- ◆ Utilize **count plots** to show the number of students enrolled in each course.
- ◆ Apply **box plots** to analyze variations in student ratings across courses.
- ◆ Implement **violin plots** to study the distribution of student enrollment by course difficulty levels (Beginner, Intermediate, Advanced).
- ◆ Use **swarm plots** to visualize individual student enrollments for each course category.
- ◆ Customize colors and themes to improve readability and presentation.

## Mini Project 2: Supermarket Sales Analysis

**Objective:** Examine sales patterns, customer purchasing behavior, and product pricing using Seaborn's categorical data visualization.

- ◆ Use **bar plots** to compare the average revenue generated by different product categories (e.g., Electronics, Groceries, Clothing).
- ◆ Utilize **count plots** to visualize the number of purchases made per product category.
- ◆ Apply **box plots** to identify variations in product prices across categories.
- ◆ Implement **violin plots** to study the distribution of product discounts by category.
- ◆ Use **swarm plots** to display individual product sales within each category.
- ◆ Customize colors and styles to make the visualizations more insightful.

Would you like step-by-step implementation for these projects? 

## Day 63

### Distribution Plots (Histograms & KDE)

In data visualization, distribution plots are used to understand the spread and pattern of data. These plots give insights into how data points are distributed across different ranges. The two most common types of distribution plots are **Histograms** and **Kernel Density Estimation (KDE)** plots. Seaborn provides several tools to visualize these distributions.

## 1. Histograms & KDE Plots

A **histogram** is a representation of the distribution of a dataset, where the data is divided into bins, and the frequency of data points in each bin is represented as bars. A **KDE plot** is a smoothed, continuous version of a histogram, representing the probability density function of a continuous random variable.

Syntax:

```
# Histogram  
sns.histplot(data, kde=False, bins=30)
```

```
# KDE Plot  
sns.kdeplot(data)
```

Explanation:

- **sns.histplot()**: Creates a histogram. You can also overlay a KDE plot using `kde=True`.
- **sns.kdeplot()**: Creates a smooth curve over the data representing the estimated probability density function.

Example:

```
import seaborn as sns  
import matplotlib.pyplot as plt  
import numpy as np  
  
# Generate random data  
data = np.random.normal(loc=0, scale=1, size=1000)
```

```
# Histogram  
sns.histplot(data, kde=False, bins=30) # Plotting Histogram  
plt.title('Histogram of Data')  
plt.show()
```

```
# KDE Plot  
sns.kdeplot(data)  
plt.title('KDE Plot of Data')  
plt.show()
```

In the above example:

- **Histogram:** We plot the data as a histogram with 30 bins.
- **KDE Plot:** A smooth curve representing the density of the data.

## 2. Dist Plot

`displot()` is a high-level function in Seaborn that combines both histograms and KDE plots into one. It is useful for displaying both distributions and density estimates.

Syntax:

```
sns.displot(data, kde=True, bins=30)
```

Explanation:

- **kde=True:** Overlay a KDE plot on the histogram.
- **bins:** Control the number of bins in the histogram.

Example:

```
sns.displot(data, kde=True, bins=30) # Both histogram and KDE  
plt.title('Histogram & KDE Combined')  
plt.show()
```

### 3. Customizing Bin Sizes in Histograms

In histograms, the number of bins affects how data is grouped. More bins show finer detail, but too many can make the plot noisy. Fewer bins make the distribution less detailed.

Syntax:

```
sns.histplot(data, bins=50)
```

Example:

```
# Histogram with 50 bins  
sns.histplot(data, bins=50)  
plt.title('Histogram with Custom Bin Size')  
plt.show()
```

In this example, we have customized the bin size to 50 to provide a finer granularity of the data distribution.

## 4. Adding Multiple KDE Lines for Comparison

KDE plots can be used to compare the distributions of different data sets by overlaying multiple KDE lines. This is useful for comparing the distributions of two or more datasets on the same plot.

Syntax:

```
sns.kdeplot(data1, label='Dataset 1')
sns.kdeplot(data2, label='Dataset 2')
```

Example:

```
# Generate two sets of data
data1 = np.random.normal(loc=0, scale=1, size=1000)
data2 = np.random.normal(loc=2, scale=1, size=1000)
```

```
# Plot KDE lines for both datasets
sns.kdeplot(data1, label='Dataset 1')
sns.kdeplot(data2, label='Dataset 2')
```

```
plt.title('Comparing Two KDE Plots')
plt.legend()
plt.show()
```

In this example, we plot the KDEs of two datasets, one centered at 0 and the other at 2. By overlaying these two KDE lines, we can compare their distributions.

## 5. Using Hue for Categorization

The **hue** parameter allows us to categorize data by a categorical variable, and it colors the plot differently based on the unique values of that variable. This can be used to group data in histograms, KDE plots, and other types of plots.

Syntax:

```
sns.histplot(data, hue='category')  
sns.kdeplot(data, hue='category')
```

Example:

```
# Load a built-in dataset (e.g., Titanic dataset)  
titanic = sns.load_dataset("titanic")  
  
# Create a histogram with hue (Categorizing by 'sex')  
sns.histplot(titanic, x="age", hue="sex", kde=True)  
plt.title('Age Distribution by Gender')  
plt.show()
```

In this example, we use the hue parameter to categorize the age distribution of passengers by gender (male and female) from the Titanic dataset. Each gender is represented by a different color in the plot.

## Summary of Functions and Their Use Cases:

### 1. Histograms (sns.histplot()):

- a. Useful for showing the frequency of data points within specified ranges (bins).
- b. Can be customized with the number of bins and whether to include a KDE curve.

**2. KDE Plot (sns.kdeplot()):**

- a. Useful for estimating the probability density function (PDF) of a continuous variable.
- b. Adds smooth curves to the data to show distribution.

**3. Dist Plot (sns.distplot()):**

- a. A higher-level function that combines histograms and KDE plots, making it easier to visualize the distribution and density together.

**4. Customizing Bin Sizes:**

- a. Helps adjust the number of bins in the histogram, allowing control over the granularity of the data representation.

**5. Multiple KDE Lines:**

- a. Useful for comparing the distribution of two or more datasets. You can overlay multiple KDE plots to visualize different groups' distributions.

**6. Hue for Categorization:**

- a. Use the **hue** parameter to categorize data by a categorical variable, helping to distinguish between different groups in a dataset.

**Conclusion:**

Distribution plots like histograms and KDEs are essential tools in data visualization. They provide valuable insights into the spread and distribution of data. By using Seaborn, you can easily customize these plots, compare multiple datasets, and categorize data to make your visualizations more insightful and informative.

## Mini Project 1: Analyzing Sales Data Distribution

In this project, we will analyze a dataset of sales transactions and visualize the distribution of sales amounts using histograms and KDE plots. Additionally, we will explore how the distribution varies across different regions using the **hue** parameter.

Project Tasks:

1. Import necessary libraries.
2. Load the sales data (CSV format) containing columns like SalesAmount, Region, and Date.
3. Check for missing values and handle them.
4. Create a basic histogram of SalesAmount.
5. Overlay a KDE plot on the histogram to visualize the sales distribution.
6. Customize the bin sizes in the histogram to find the optimal number of bins.
7. Use the **hue** parameter to visualize the distribution of sales across different regions (categorize by Region).
8. Create a dist plot (sns.distplot()) that includes both the histogram and KDE plot.
9. Add multiple KDE lines for comparison (compare sales distribution between two different regions).
10. Adjust the axis labels and titles for clarity.
11. Customize colors of the plots to match the region categories.
12. Save the figure as a PNG file.
13. Display the final visualizations.

Step-by-Step Implementation:

```
# Task 1: Import necessary libraries  
import seaborn as sns  
import matplotlib.pyplot as plt
```

```
import pandas as pd

# Task 2: Load sales data (replace with actual file path)
sales_data = pd.read_csv("sales_data.csv")

# Task 3: Check for missing values and handle them
print(sales_data.isnull().sum()) # Check for missing values
sales_data.dropna(inplace=True) # Drop rows with missing values

# Task 4: Create a basic histogram of SalesAmount
sns.histplot(sales_data['SalesAmount'], kde=False, bins=30)
plt.title('Sales Amount Distribution')
plt.xlabel('Sales Amount')
plt.ylabel('Frequency')
plt.show()

# Task 5: Overlay a KDE plot on the histogram
sns.histplot(sales_data['SalesAmount'], kde=True, bins=30)
plt.title('Sales Amount Distribution with KDE')
plt.xlabel('Sales Amount')
plt.ylabel('Frequency')
plt.show()

# Task 6: Customize the bin sizes in the histogram
sns.histplot(sales_data['SalesAmount'], kde=True, bins=50)
plt.title('Sales Amount Distribution with Custom Bin Size')
plt.xlabel('Sales Amount')
plt.ylabel('Frequency')
plt.show()

# Task 7: Use hue for categorization (visualizing by Region)
```

```
sns.histplot(sales_data, x='SalesAmount', hue='Region', kde=True, bins=30)
plt.title('Sales Amount Distribution by Region')
plt.xlabel('Sales Amount')
plt.ylabel('Frequency')
plt.show()

# Task 8: Create a dist plot with both histogram and KDE plot
sns.displot(sales_data['SalesAmount'], kde=True, bins=30)
plt.title('Sales Amount Distribution with Dist Plot')
plt.xlabel('Sales Amount')
plt.ylabel('Frequency')
plt.show()

# Task 9: Adding multiple KDE lines for comparison (by Region)
sns.kdeplot(sales_data[sales_data['Region'] == 'North']['SalesAmount'],
label='North', color='blue')
sns.kdeplot(sales_data[sales_data['Region'] == 'South']['SalesAmount'],
label='South', color='red')
plt.title('Sales Amount KDE Comparison by Region')
plt.xlabel('Sales Amount')
plt.ylabel('Density')
plt.legend()
plt.show()

# Task 10: Adjust the axis labels and titles for clarity
sns.histplot(sales_data['SalesAmount'], kde=True, bins=30)
plt.title('Sales Amount Distribution with KDE')
plt.xlabel('Sales Amount in USD')
plt.ylabel('Frequency')
plt.show()
```

```
# Task 11: Customize the colors for different regions in the hue parameter
sns.histplot(sales_data, x='SalesAmount', hue='Region', kde=True, bins=30,
palette='Set1')
plt.title('Sales Amount Distribution by Region (Customized Colors)')
plt.xlabel('Sales Amount')
plt.ylabel('Frequency')
plt.show()

# Task 12: Save the figure as a PNG file
sns.histplot(sales_data['SalesAmount'], kde=True, bins=30)
plt.title('Sales Amount Distribution with KDE')
plt.xlabel('Sales Amount')
plt.ylabel('Frequency')
plt.savefig('sales_amount_distribution.png')

# Task 13: Display the final visualization (already displayed during the process)
plt.show()
```

## Mini Project 2: Exploring Students' Exam Scores

In this project, we will analyze a dataset of students' exam scores and visualize their score distribution using histograms and KDE plots. We will explore how the distribution changes based on whether the student passed or failed.

Project Tasks:

1. Import necessary libraries.
2. Load the student exam data (CSV format) with columns like Score and Passed.
3. Handle any missing data in the dataset.

4. Plot a histogram of the exam scores (Score).
5. Overlay a KDE plot on the histogram to better visualize the distribution.
6. Customize the bin size of the histogram.
7. Use the **hue** parameter to categorize by Passed (passed vs failed).
8. Create a dist plot (sns.distplot()) combining histogram and KDE.
9. Compare the distribution of scores between passed and failed students using multiple KDE lines.
10. Add axis labels and a title to the plots.
11. Customize the colors for the passed and failed categories.
12. Save the plot as an image.
13. Display the final plots.

#### **Step-by-Step Implementation:**

```
# Task 1: Import necessary libraries
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

# Task 2: Load student exam data (replace with actual file path)
exam_data = pd.read_csv("student_exam_data.csv")

# Task 3: Handle missing data
print(exam_data.isnull().sum()) # Check for missing values
exam_data.dropna(inplace=True) # Drop rows with missing values

# Task 4: Plot a basic histogram of exam scores
sns.histplot(exam_data['Score'], kde=False, bins=20)
plt.title('Distribution of Exam Scores')
plt.xlabel('Score')
plt.ylabel('Frequency')
```

```
plt.show()
```

```
# Task 5: Overlay a KDE plot on the histogram  
sns.histplot(exam_data['Score'], kde=True, bins=20)  
plt.title('Distribution of Exam Scores with KDE')  
plt.xlabel('Score')  
plt.ylabel('Frequency')  
plt.show()
```

```
# Task 6: Customize the bin size in the histogram  
sns.histplot(exam_data['Score'], kde=True, bins=50)  
plt.title('Distribution of Exam Scores with Custom Bin Size')  
plt.xlabel('Score')  
plt.ylabel('Frequency')  
plt.show()
```

```
# Task 7: Use hue for categorizing by Passed/Failed  
sns.histplot(exam_data, x='Score', hue='Passed', kde=True, bins=20)  
plt.title('Distribution of Exam Scores (Passed vs Failed)')  
plt.xlabel('Score')  
plt.ylabel('Frequency')  
plt.show()
```

```
# Task 8: Create a dist plot combining histogram and KDE  
sns.displot(exam_data['Score'], kde=True, bins=20)  
plt.title('Distribution of Exam Scores with Dist Plot')  
plt.xlabel('Score')  
plt.ylabel('Frequency')  
plt.show()
```

```
# Task 9: Add multiple KDE lines for Passed vs Failed
```

```
sns.kdeplot(exam_data[exam_data['Passed'] == 'Yes']['Score'], label='Passed',
color='green')
sns.kdeplot(exam_data[exam_data['Passed'] == 'No']['Score'], label='Failed',
color='red')
plt.title('Exam Scores Distribution for Passed and Failed Students')
plt.xlabel('Score')
plt.ylabel('Density')
plt.legend()
plt.show()

# Task 10: Add axis labels and title
sns.histplot(exam_data['Score'], kde=True, bins=20)
plt.title('Exam Score Distribution with KDE')
plt.xlabel('Exam Score')
plt.ylabel('Frequency')
plt.show()

# Task 11: Customize colors for Passed/Failed categories
sns.histplot(exam_data, x='Score', hue='Passed', kde=True, bins=20,
palette='coolwarm')
plt.title('Distribution of Exam Scores with Custom Colors')
plt.xlabel('Score')
plt.ylabel('Frequency')
plt.show()

# Task 12: Save the plot as a PNG image
sns.histplot(exam_data['Score'], kde=True, bins=20)
plt.title('Exam Score Distribution with KDE')
plt.xlabel('Score')
plt.ylabel('Frequency')
plt.savefig('exam_score_distribution.png')
```

```
# Task 13: Display the final plot (already displayed during the process)
plt.show()
```

### Conclusion:

These two mini projects showcase how to work with **distribution plots** like histograms and KDE in Seaborn. You can customize these plots, categorize data using the **hue** parameter, and compare multiple groups (e.g., different regions or passed/failed students). Understanding these concepts will help you visualize the distribution of your data effectively, aiding in better decision-making.

## Mini Project: Analyzing the Distribution of Employee Salaries

In this project, we will analyze a dataset containing information about employee salaries, and visualize the salary distribution using histograms and KDE plots. We will also categorize the data based on departments and compare salary distributions across different departments.

### Project Tasks:

1. Import necessary libraries (Seaborn, Matplotlib, Pandas).
2. Load the employee salary data (CSV format) containing columns like Salary, Department, and Experience.
3. Check for missing values in the dataset and handle them (drop or fill).
4. Create a basic histogram of the Salary column.
5. Overlay a KDE plot on the histogram to visualize the salary distribution more smoothly.
6. Customize the bin sizes in the histogram to find the optimal bin size for better insights.

7. Use the **hue** parameter to visualize salary distribution categorized by Department.
8. Create a dist plot (sns.distplot()) combining the histogram and KDE plot for a better understanding of the salary distribution.
9. Add multiple KDE lines for comparing salary distributions between two or more departments.
10. Adjust the axis labels (e.g., Salary in USD) and add a title for better clarity.
11. Customize the colors of the plots based on the departments using a specific color palette.
12. Save the final visualization as a PNG file for sharing or reporting.
13. Display the final visualizations showing salary distribution and comparisons across different departments.

These tasks will guide you through analyzing and visualizing salary data and comparing it across different categories like departments, helping you gain insights into how salaries are distributed within an organization.

## **Mini Project 1: Visualizing the Age Distribution of Customers in an E-commerce Platform**

In this project, we will analyze and visualize the age distribution of customers on an e-commerce platform. We will use histograms, KDE plots, and the hue parameter to categorize the data by customer gender.

Project Tasks:

- Task 1:** Import necessary libraries (Seaborn, Matplotlib, Pandas).
- Task 2:** Load the customer data, which includes columns such as Age, Gender, Purchase Amount.
- Task 3:** Handle any missing values or outliers in the dataset.
- Task 4:** Create a histogram for the Age column to visualize the age distribution of all customers.

- Task 5:** Add a KDE plot to the histogram to create a smoother distribution curve of age.
- Task 6:** Adjust the number of bins in the histogram to achieve better clarity.
- Task 7:** Use the **hue** parameter to categorize the age distribution by gender.
- Task 8:** Create a `sns.displot()` to combine the histogram and KDE plot with a faceted grid.
- Task 9:** Add multiple KDE lines for comparing age distributions between male and female customers.
- Task 10:** Customize axis labels and the title to make the visualization more informative.
- Task 11:** Use a color palette to differentiate between male and female customers in the plot.
- Task 12:** Save the final plot as an image file for reporting purposes.
- Task 13:** Display the final visualization showing the age distribution by gender.

## **Mini Project 2: Analyzing Exam Scores of Students in Different Subjects**

In this project, we will analyze and visualize the exam scores of students in different subjects. The data will contain columns such as Math, Science, English, and Gender. We will visualize the score distributions using histograms and KDE plots and compare the distributions for each subject.

Project Tasks:

- Task 1:** Import necessary libraries (Seaborn, Matplotlib, Pandas).
- Task 2:** Load the dataset containing the exam scores for Math, Science, and English subjects along with the Gender column.
- Task 3:** Check and handle missing values or any irregularities in the dataset.

**Task 4:** Create histograms to visualize the score distributions for each subject (Math, Science, English).

**Task 5:** Add KDE plots to the histograms to visualize smoother distributions of scores for each subject.

**Task 6:** Customize the bin sizes for the histograms to get the most meaningful view of the distributions.

**Task 7:** Use the **hue** parameter to visualize score distributions categorized by gender.

**Task 8:** Create a `sns.displot()` to combine the histogram and KDE for each subject and gender.

**Task 9:** Add multiple KDE lines for comparing the score distributions across different genders.

**Task 10:** Adjust the axis labels to clearly indicate the subjects and scores.

**Task 11:** Use a custom color palette for the plots to distinguish between male and female students.

**Task 12:** Save the plot as an image file for later use.

**Task 13:** Display the final plot showing the score distributions across subjects and genders.

These mini projects will help you to practice visualizing categorical data using histograms and KDE plots, as well as comparing distributions based on categories (like gender).

# Day 64

## Correlation & Heatmaps

What is Correlation? Why is it Important?

**Correlation** refers to the statistical relationship between two variables. If two variables are correlated, it means that they have a tendency to move together, i.e., when one variable changes, the other tends to change in a predictable way.

Correlation helps to understand how two variables are related to each other, and it's important in the following cases:

- **Predictive modeling:** Correlation helps us understand the relationship between features, which can be useful for predicting future data points.
- **Data cleaning:** By checking correlations, you can identify redundant features or features that are highly related.
- **Feature engineering:** Correlation can guide the creation of new features that might provide better insights.

### Types of Correlation:

- **Positive Correlation:** When one variable increases, the other also increases.
- **Negative Correlation:** When one variable increases, the other decreases.
- **No Correlation:** When one variable changes, there is no predictable change in the other variable.

Creating a Correlation Matrix using Pandas

A **correlation matrix** is a table that shows the correlation coefficients between many variables. Each cell in the matrix represents the correlation between two

variables. A value of +1 indicates a perfect positive correlation, -1 indicates a perfect negative correlation, and 0 indicates no correlation.

### Syntax:

```
import pandas as pd
```

```
# Assuming `df` is your DataFrame  
correlation_matrix = df.corr()  
print(correlation_matrix)
```

In this example:

- `df.corr()` calculates the correlation matrix for all numeric columns in the DataFrame.
- This matrix helps us identify which features are correlated and to what extent.

### Creating a Heatmap (`sns.heatmap()`)

A **heatmap** is a data visualization technique that uses color to represent values in a matrix, making it easier to identify patterns or correlations. It's commonly used to visualize the correlation matrix.

### Syntax:

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

```
# Create a heatmap from a correlation matrix  
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', linewidths=0.5)  
plt.show()
```

**Parameters:**

- `annot=True`: Annotates each cell with the numeric value of the correlation.
- `cmap='coolwarm'`: Specifies the color palette (cool for negative correlations, warm for positive).
- `linewidths=0.5`: Adds a border around each cell to improve readability.

**Customizing Heatmaps (Color Maps, Annotations, Line Widths)**

Heatmaps are highly customizable. You can change the color map (`cmap`), adjust the annotations (`annot`), or even adjust the line width between cells (`linewidths`).

**Example:**

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

# Sample DataFrame
data = {
    'Math': [88, 92, 80, 89, 78],
    'English': [85, 87, 82, 91, 76],
    'Science': [90, 92, 88, 95, 80]
}
df = pd.DataFrame(data)

# Compute correlation matrix
correlation_matrix = df.corr()

# Create a heatmap with customizations
sns.heatmap(correlation_matrix, annot=True, cmap='Blues', linewidths=0.5,
            linecolor='black', fmt='.2f')
plt.title('Correlation Matrix of Scores')
```

```
plt.show()
```

### Customizations:

- `cmap='Blues'`: Uses the blue color scale for the heatmap.
- `linewidths=0.5`: Sets the width of the lines separating cells.
- `linecolor='black'`: Sets the color of the line dividing cells.
- `fmt='.2f'`: Displays the correlation values with 2 decimal points.

### Handling Missing Data in Heatmaps

When creating heatmaps, missing values (`NaN`) can be problematic. You can handle missing data by:

1. **Filling missing values**: Use techniques like mean imputation, forward fill, or backward fill to replace `NaN` values.
2. **Masking missing values**: You can use a mask to hide `NaN` values in the heatmap.

### Example of Filling Missing Data:

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

# Sample DataFrame with missing values
data = {
    'Math': [88, 92, None, 89, 78],
    'English': [85, None, 82, 91, 76],
    'Science': [None, 92, 88, 95, 80]
}
df = pd.DataFrame(data)
```

```
# Fill missing values with the mean of each column  
df.fillna(df.mean(), inplace=True)  
  
# Compute correlation matrix  
correlation_matrix = df.corr()  
  
# Create heatmap with missing values filled  
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', linewidths=0.5)  
plt.title('Correlation Matrix with Missing Values Handled')  
plt.show()
```

### Example of Masking Missing Data:

```
# Masking missing values in the correlation matrix  
mask = df.isnull()  
  
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', linewidths=0.5,  
mask=mask)  
plt.title('Correlation Matrix with Missing Data Masked')  
plt.show()
```

Here:

- `df.fillna(df.mean(), inplace=True)` fills missing values with the mean of each column.
- `mask = df.isnull()` creates a mask to hide missing data.

## Summary

- **Correlation** is the statistical relationship between variables. It helps in identifying patterns and understanding data.
- A **correlation matrix** is a table that shows the correlation values between pairs of variables.
- A **heatmap** is a graphical representation of data where individual values are represented by colors.
- Customizing heatmaps allows you to improve the readability and aesthetics of your plots.
- **Missing data** in heatmaps can be handled either by filling or masking the missing values to ensure the heatmap is generated correctly.

With these steps, you can visualize correlations between different variables in your dataset and interpret the relationships effectively.

Here are two real-life mini project requirements related to **Correlation & Heatmaps** with step-by-step implementation and detailed code explanation:

## Mini Project 1: Analyzing Student Scores - Correlation & Heatmap

### Project Objective:

You are provided with student scores in multiple subjects, and you need to analyze the correlation between the scores of different subjects and visualize the correlation matrix using a heatmap.

**Steps:**

- 1. Create a DataFrame with Sample Data**
2. You have the scores of students in three subjects: Math, English, and Science.
- 3. Create the Correlation Matrix**

Calculate the correlation between the subjects (Math, English, Science).

- 4. Generate a Heatmap**

Create a heatmap to visualize the correlation matrix.

- 5. Customize the Heatmap**

Apply custom color maps and add annotations to the heatmap for better readability.

- 6. Handle Missing Data**

If there are any missing values, use imputation or a mask to handle them before plotting.

**Step-by-Step Implementation:**

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

# Step 1: Create a DataFrame with Student Scores
data = {
```

```
'Math': [88, 92, 80, 89, 78],  
'English': [85, 87, 82, 91, 76],  
'Science': [90, 92, 88, 95, 80]  
}  
df = pd.DataFrame(data)  
  
# Step 2: Create the Correlation Matrix using pandas  
correlation_matrix = df.corr()  
print("Correlation Matrix:\n", correlation_matrix)  
  
# Step 3: Generate a Heatmap using seaborn  
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', linewidths=0.5)  
plt.title('Correlation Matrix of Student Scores')  
plt.show()  
  
# Step 4: Customize the Heatmap  
sns.heatmap(correlation_matrix, annot=True, cmap='Blues', linewidths=1,  
linecolor='black', fmt='.2f')  
plt.title('Customized Correlation Matrix of Student Scores')  
plt.show()  
  
# Step 5: Handle Missing Data (imputing with mean if necessary)  
df['Math'][2] = None # Simulating a missing value in the Math column  
df.fillna(df.mean(), inplace=True) # Imputing missing values with column mean  
  
# Recompute and plot the heatmap after handling missing data  
correlation_matrix = df.corr()  
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', linewidths=0.5)  
plt.title('Correlation Matrix After Handling Missing Data')  
plt.show()
```

## Explanation:

### 1. Creating DataFrame:

- a. data: A dictionary where keys are subjects, and values are lists of student scores.
- b. df: Converts the dictionary into a Pandas DataFrame.

### 2. Creating Correlation Matrix:

- a. df.corr(): Computes the correlation matrix between the columns in the DataFrame.

### 3. Generating Heatmap:

- a. sns.heatmap(): Plots the correlation matrix as a heatmap. We set annot=True to show the correlation values in each cell and cmap='coolwarm' for a color gradient.

### 4. Customizing Heatmap:

- a. The color palette is changed to Blues for aesthetic purposes, and linewidths=1 and linecolor='black' help distinguish cells clearly.

### 5. Handling Missing Data:

- a. A missing value is simulated in the 'Math' column.
- b. df.fillna(df.mean(), inplace=True) fills the missing values with the mean of the respective column.

## Mini Project 2: Sales Data Analysis - Correlation & Heatmap

### Project Objective:

You have sales data from different stores and need to analyze and visualize the relationship between sales and promotions across different regions.

**Steps:****1. Create a DataFrame with Sales Data**

You have sales data for stores, including Sales, Promotions, and Store Location.

**2. Create the Correlation Matrix**

Calculate the correlation matrix between Sales, Promotions, and Store Location.

**3. Generate a Heatmap**

Use seaborn to plot a heatmap of the correlation matrix.

**4. Customize the Heatmap**

Adjust colors and formatting for better visualization.

**5. Handle Missing Data**

Handle any missing data by filling or masking it.

**Step-by-Step Implementation:**

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

# Step 1: Create DataFrame with Sales Data
sales_data = {
    'Sales': [1000, 1500, 1200, 1300, 1100],
    'Promotions': [200, 250, 150, 230, 210],
    'Store_Location': ['East', 'West', 'North', 'South', 'East']}
```

```
}

df = pd.DataFrame(sales_data)

# Step 2: Create the Correlation Matrix
# First, we convert Store_Location into numeric values for correlation
computation
df['Store_Location'] = df['Store_Location'].map({'East': 1, 'West': 2, 'North': 3,
'South': 4})
correlation_matrix = df.corr()
print("Correlation Matrix:\n", correlation_matrix)

# Step 3: Generate a Heatmap using seaborn
sns.heatmap(correlation_matrix, annot=True, cmap='viridis', linewidths=0.5)
plt.title('Correlation Matrix of Sales and Promotions')
plt.show()

# Step 4: Customize the Heatmap
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', linewidths=0.5,
fmt='.2f')
plt.title('Customized Correlation Matrix for Sales Data')
plt.show()

# Step 5: Handle Missing Data (if any)
df['Sales'][2] = None # Simulate missing data in Sales
df.fillna(df.mean(), inplace=True) # Impute missing values with the mean of the
column

# Recompute the correlation matrix after filling missing values
correlation_matrix = df.corr()

# Plot the heatmap again after handling missing data
```

```
sns.heatmap(correlation_matrix, annot=True, cmap='Blues', linewidths=0.5)
plt.title('Correlation Matrix After Handling Missing Data')
plt.show()
```

## Explanation:

### 1. Creating DataFrame:

- a. sales\_data: A dictionary containing sales information such as Sales, Promotions, and Store Location.
- b. df: Converts the dictionary into a Pandas DataFrame.

### 2. Creating Correlation Matrix:

- a. df.corr(): Calculates the correlation matrix of numeric columns (Sales, Promotions, and Store\_Location).

### 3. Generating Heatmap:

- a. sns.heatmap(): Creates the heatmap with the correlation matrix. Customizes color map (viridis and coolwarm) for better visibility.

### 4. Customizing Heatmap:

- a. Adjusted colors and added annotations with annot=True to display correlation values on the heatmap cells.

### 5. Handling Missing Data:

- a. Missing values in the 'Sales' column are handled using df.fillna(df.mean(), inplace=True), which fills missing values with the mean of each column.

## Summary:

1. **Correlation Analysis:** This helps us understand the relationship between variables in the dataset. It's useful for identifying patterns that can guide decision-making.
2. **Heatmap Visualization:** A heatmap makes it easy to visualize the correlation matrix, where the color gradient quickly reveals strong positive or negative correlations.
3. **Missing Data Handling:** Ensuring that missing values are addressed appropriately before visualizing data is essential for accurate insights.

By implementing these mini projects, you'll gain hands-on experience in using correlation analysis and visualizations like heatmaps to extract insights from real-world datasets.

Here's a real-life mini project related to **Correlation & Heatmaps** with 13 tasks for you to implement:

### **Mini Project: Financial Data Analysis - Correlation & Heatmap**

#### **Project Objective:**

Analyze financial data (such as stock prices, revenue, and expenses) from multiple companies to understand the relationships between these variables and visualize the correlation matrix using heatmaps.

## **13 Tasks:**

### **1. Dataset Collection**

Collect or create a dataset with financial data of different companies (e.g., stock prices, revenue, expenses, profit margin, etc.).

### **2. Data Preprocessing**

Clean the dataset by handling missing data and ensuring that all columns have appropriate data types for correlation analysis.

### **3. Check for Missing Values**

Identify columns with missing values, and summarize the total number of missing values in each column.

### **4. Impute or Drop Missing Values**

Decide whether to drop rows with missing values or fill them with appropriate values (mean, median, or other imputation techniques).

### **5. Calculate Correlation Matrix**

Use pandas.corr() to calculate the correlation matrix between the financial metrics in the dataset.

### **6. Display the Correlation Matrix**

Print the correlation matrix in a readable format to check the relationships between variables.

### **7. Create a Heatmap for Correlation Matrix**

Use sns.heatmap() to create a heatmap of the correlation matrix to visualize the relationships between the financial variables.

## **8. Customize the Heatmap Colors**

Use different color maps (e.g., coolwarm, viridis, etc.) to customize the heatmap for better clarity and aesthetic appeal.

## **9. Add Annotations to Heatmap**

Annotate the heatmap with the correlation values inside the cells to make the relationships clearer.

## **10. Adjust Line Widths and Cell Borders**

Experiment with linewidths and linecolor parameters in sns.heatmap() to adjust the line thickness and cell borders.

## **11. Handle Missing Data in the Heatmap**

Ensure that missing data in the correlation matrix is handled correctly, either by masking or imputation, before visualizing the heatmap.

## **12. Use a Custom Color Map**

Create and apply a custom color map to the heatmap to better visualize positive and negative correlations.

## **13. Analyze Insights from the Heatmap**

Analyze the heatmap for insights. Look for strong positive or negative correlations between financial variables, and prepare a short report summarizing the findings.

## Example Dataset (for context):

Company Name	Stock Price	Revenue	Expense	Profit Margin
Company A	120	2000	1500	0.25
Company B	150	2500	2000	0.20
Company C	100	1500	1200	0.15
Company D	130	2300	1900	0.22
Company E	140	2100	1600	0.23

### Goal:

The aim of this mini project is to identify which financial variables are most closely related and use the heatmap to present these relationships. For example, you might find that **Stock Price** has a high positive correlation with **Revenue** or **Profit Margin**. The heatmap will help visually identify these patterns for further analysis.

Here are two more real-life mini project requirements related to **Correlation & Heatmaps**:

### Mini Project 1: Healthcare Data Analysis - Correlation & Heatmap

#### Project Objective:

Analyze healthcare data (e.g., patient health metrics like blood pressure, cholesterol, age, etc.) to understand the relationships between these variables and visualize them using a heatmap to identify key trends.

## 13 Tasks:

### 1. Dataset Collection

2. Obtain a healthcare dataset (e.g., heart disease dataset, diabetes dataset) containing various health metrics such as age, cholesterol levels, blood pressure, BMI, etc.

### 3. Data Preprocessing

Clean the dataset by checking for missing values, handling outliers, and ensuring proper data types for each column.

### 4. Check for Missing Values

Identify columns that have missing values and determine the number of missing entries in each column.

### 5. Handle Missing Data

Impute or remove rows with missing values using strategies like mean imputation, median imputation, or forward/backward filling.

### 6. Calculate Correlation Matrix

Use pandas.corr() to compute the correlation matrix of the health metrics in the dataset.

### 7. Visualize the Correlation Matrix

Display the correlation matrix in a readable format to explore the relationships between the health variables.

### 8. Create a Heatmap

Generate a heatmap of the correlation matrix using sns.heatmap() to visually inspect the correlations.

### **9. Customize Heatmap Color Map**

Customize the color map to a suitable one (e.g., YIGnBu, coolwarm) to enhance the heatmap's clarity.

### **10. Add Annotations to Heatmap**

Use the annot=True parameter to annotate the heatmap with numerical values of the correlation coefficients.

### **11. Adjust Line Widths in Heatmap**

Adjust the line widths and colors between cells using the linewidths and linecolor parameters for better visualization.

### **12. Handle Missing Data in Heatmap**

Ensure that missing or NaN values in the correlation matrix are handled properly, possibly using masking techniques.

### **13. Create Custom Color Map**

Design and apply a custom color map to the heatmap for better visual representation of health metric correlations.

### **14. Draw Conclusions from Heatmap**

Based on the heatmap, analyze which health metrics are most correlated with each other and create a summary report of your findings.

## Mini Project 2: Environmental Data Analysis - Correlation & Heatmap

### Project Objective:

Analyze environmental data (e.g., air quality, temperature, humidity, and pollution levels) and use a heatmap to identify correlations and trends between various environmental variables.

### 13 Tasks:

#### 1. Dataset Collection

Obtain an environmental dataset with various environmental parameters like air quality index (AQI), temperature, humidity, CO2 levels, etc.

#### 2. Data Preprocessing

Clean the dataset by removing duplicate entries, checking for missing data, and ensuring correct data types for all columns.

#### 3. Identify Missing Values

Analyze the dataset to identify missing values, focusing on columns such as temperature, humidity, and pollution levels.

#### 4. Handle Missing Data

Implement strategies for handling missing values, either by filling them with mean/median values or removing rows with missing data.

#### 5. Compute the Correlation Matrix

Use pandas.corr() to compute the correlation matrix for the environmental parameters in the dataset.

## **6. Display the Correlation Matrix**

Display the correlation matrix in tabular form to easily observe how each environmental factor correlates with others.

## **7. Generate a Heatmap**

Create a heatmap using sns.heatmap() to visually inspect the correlations between the environmental variables.

## **8. Customize the Heatmap Colors**

Experiment with different color maps such as viridis, coolwarm, or inferno to make the heatmap more informative.

## **9. Add Annotations to Heatmap**

Enable annotations (annot=True) to display the correlation values inside each cell of the heatmap for easier interpretation.

## **10. Refine Heatmap with Line Widths**

Adjust the linewidths and linecolor properties of the heatmap to customize the appearance and borders between cells.

## **11. Mask Missing Data in Heatmap**

If there are any missing or NaN values in the correlation matrix, use masking to hide them in the heatmap for clearer visualization.

## **12. Use a Custom Color Map**

Design a custom color map suitable for environmental data to differentiate positive and negative correlations more clearly.

### 13. Interpret Results from Heatmap

Analyze the heatmap for correlations between environmental factors (e.g., pollution levels and temperature) and summarize your findings in a report.

Both projects will help you gain practical experience in working with correlation analysis, creating heatmaps, handling missing data, and interpreting relationships between different variables in real-world datasets.

## Day 65

### Pair Plots & Joint Plots

Pair plots and joint plots are useful visualization tools when working with multiple variables in a dataset, especially for understanding relationships between different features.

#### 1. Pair Plots (`sns.pairplot()`)

##### Definition:

A pair plot is a grid of scatter plots showing the pairwise relationships between the features in a dataset. It helps in visualizing how variables correlate with each other. Pair plots are particularly useful for exploring high-dimensional datasets and identifying trends, patterns, or correlations.

##### Syntax:

```
sns.pairplot(data, hue=None, palette=None, kind='scatter', markers=None,  
height=2.5)
```

**Parameters:**

- data: The DataFrame containing the data.
- hue: Variable name (string) that will be used for color encoding (categorical).
- palette: Color palette for categorical variables.
- kind: Type of plot (scatter, kde, reg). Default is scatter.
- markers: Marker type for the scatter plot.
- height: The size of each plot (in inches).

**Example of Pair Plot:**

Let's say we have a dataset containing information about different species of flowers (Iris dataset), and we want to visualize how the features (sepal length, sepal width, petal length, petal width) correlate with each other.

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

# Load the Iris dataset
iris = sns.load_dataset('iris')

# Create a pair plot with hue set to the species column
sns.pairplot(iris, hue='species', palette='Set2', kind='scatter')

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

**Explanation:**

- The pairplot() function creates a grid of scatter plots for all pairs of variables in the Iris dataset.
- The hue='species' parameter colors the points based on the species of flowers, helping us distinguish between them.
- kind='scatter' means we're plotting scatter plots by default.
- palette='Set2' specifies the color palette used for the species.

**Output:** You'll see a grid of scatter plots with each pair of features (e.g., sepal length vs petal length), and points are colored by the species.

## 2. Customizing Pair Plots

**Definition:**

You can customize pair plots by changing the visualization style, type of plot, colors, or even including regression lines.

**Customizations:**

- kind='reg': Adds a regression line.
- kind='kde': Uses kernel density estimation (KDE) instead of scatter plots.
- hue='variable': Colors the points based on a categorical variable (like species).
- height=3: Adjust the size of each individual plot.

**Example of Customizing Pair Plots:**

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

```
# Load the Iris dataset
iris = sns.load_dataset('iris')

# Create a pair plot with regression lines and hue
sns.pairplot(iris, hue='species', kind='reg', height=3, palette='coolwarm')

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

#### **Explanation:**

- kind='reg' adds a regression line for each scatter plot.
- height=3 makes each plot a bit larger for better visibility.
- palette='coolwarm' changes the color scheme to a "cool-to-warm" gradient for the species.

### **3. Joint Plots (sns.jointplot())**

#### **Definition:**

A joint plot combines scatter plots with histograms (or kernel density plots) for a pair of variables. It gives a clearer picture of the bivariate distribution, helping to understand how two variables relate.

#### **Syntax:**

```
sns.jointplot(x=None, y=None, data=None, kind='scatter', hue=None,
palette=None, height=6)
```

**Parameters:**

- x and y: The variables to be plotted.
- data: The DataFrame containing the data.
- kind: Type of plot (scatter, hex, kde, reg). Default is scatter.
- hue: Variable name that will be used for color encoding.
- palette: Color palette for categorical variables.
- height: Size of the joint plot.

**Example of Joint Plot (Scatter Plot + Histograms):**

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

# Load the Iris dataset
iris = sns.load_dataset('iris')

# Create a joint plot for sepal length vs sepal width
sns.jointplot(x='sepal_length', y='sepal_width', data=iris, kind='scatter',
hue='species')

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

**Explanation:**

- The jointplot() function creates a scatter plot for sepal length vs sepal width.
- It also includes histograms along the axes to show the distribution of the individual variables.

- hue='species' colors the points by flower species.
- The kind='scatter' parameter specifies that the plot will be a scatter plot combined with histograms.

## 4. Different Kinds of Joint Plots

### Definition:

The kind parameter in sns.jointplot() allows for different types of plots to be displayed alongside the scatter plot and histograms.

- kind='hex': Uses hexagonal binning instead of scatter.
- kind='kde': Uses kernel density estimates (KDE) for smoother plots.
- kind='reg': Adds a regression line to the scatter plot.

### Example of Hexagonal Plot (kind='hex'):

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

# Load the Iris dataset
iris = sns.load_dataset('iris')

# Create a joint plot using hexagonal binning
sns.jointplot(x='sepal_length', y='sepal_width', data=iris, kind='hex',
color='orange')

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

**Explanation:**

- kind='hex' replaces the scatter plot with hexagonal binning to show the density of points in each region.

**Example of KDE Plot (kind='kde'):**

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

# Load the Iris dataset
iris = sns.load_dataset('iris')

# Create a joint plot with KDE
sns.jointplot(x='sepal_length', y='sepal_width', data=iris, kind='kde',
color='green')

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

**Explanation:**

- kind='kde' plots kernel density estimation for the bivariate distribution, giving a smoothed version of the scatter plot.

**Example of Regression Plot (kind='reg'):**

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

```
# Load the Iris dataset
iris = sns.load_dataset('iris')

# Create a joint plot with a regression line
sns.jointplot(x='sepal_length', y='sepal_width', data=iris, kind='reg', color='blue')

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

### Explanation:

- kind='reg' adds a regression line to the scatter plot, showing the trend or relationship between sepal length and width.

### Summary of Key Points:

1. **Pair Plots:** Help visualize pairwise relationships between variables. Use sns.pairplot() to create grids of scatter plots.
2. **Joint Plots:** Combine scatter plots and histograms (or KDE) to show the relationship between two variables. Use sns.jointplot() with various kind options like scatter, hex, kde, and reg.
3. **Customization:** Both pair and joint plots can be customized with options like hue, kind, palette, and height to suit specific analysis needs.

These visualizations are incredibly useful for gaining insights into the relationships between multiple variables and are a powerful tool for exploratory data analysis.

## Real Life Mini Project 1: Visualizing the Relationship between Iris Features using Pair and Joint Plots

Project Overview:

In this project, we will explore the famous Iris dataset and use Pair and Joint plots to visualize the relationships between the features like sepal length, sepal width, petal length, and petal width. The goal is to identify patterns and correlations between these features across different Iris species.

Tasks:

### 1. Load the Iris Dataset

2. Load the Iris dataset using Seaborn and understand its structure.

### 3. Create a Pair Plot

Use sns.pairplot() to create a grid of scatter plots for all pairs of features.

### 4. Customize Pair Plot with Hue

Customize the pair plot by setting the hue parameter to color the points by species.

### 5. Change Plot Kind in Pair Plot

Modify the pair plot to display a regression line (kind='reg') instead of scatter plots.

### 6. Adjust Plot Size

Change the height parameter of the pair plot to make the plots larger or smaller.

### 7. Create a Joint Plot (Scatter + Histograms)

Use sns.jointplot() to visualize the relationship between sepal\_length and sepal\_width with scatter plots and histograms.

### **8. Customize Joint Plot with Different Kinds**

Try using different kind values (e.g., hex, kde, reg) in the joint plot to visualize the data.

### **9. Customize Joint Plot with Hue**

Add a hue parameter in the joint plot to distinguish between Iris species.

### **10. Add Regression Line in Joint Plot**

Create a joint plot with a regression line to understand the trend between the variables.

### **11. Save and Export the Plots**

Save the pair plot and joint plot as PNG images using plt.savefig().

Step-by-Step Implementation:

```
import seaborn as sns  
import matplotlib.pyplot as plt  
  
# Step 1: Load the Iris Dataset  
iris = sns.load_dataset('iris')  
  
# Step 2: Create a Pair Plot  
sns.pairplot(iris, hue='species', palette='Set2')  
plt.show()
```

```
# Step 3: Customize Pair Plot with Hue
sns.pairplot(iris, hue='species', palette='Set1', kind='scatter')
plt.show()

# Step 4: Change Plot Kind in Pair Plot (using Regression Line)
sns.pairplot(iris, hue='species', kind='reg', height=2.5)
plt.show()

# Step 5: Adjust Plot Size
sns.pairplot(iris, hue='species', height=3)
plt.show()

# Step 6: Create a Joint Plot (Scatter + Histograms)
sns.jointplot(x='sepal_length', y='sepal_width', data=iris, kind='scatter')
plt.show()

# Step 7: Customize Joint Plot with Different Kinds
sns.jointplot(x='sepal_length', y='sepal_width', data=iris, kind='hex',
color='purple')
plt.show()

# Step 8: Customize Joint Plot with Hue
sns.jointplot(x='sepal_length', y='sepal_width', data=iris, kind='kde',
hue='species')
plt.show()

# Step 9: Add Regression Line in Joint Plot
sns.jointplot(x='sepal_length', y='sepal_width', data=iris, kind='reg', color='green')
plt.show()

# Step 10: Save the plots
```

```
sns.pairplot(iris, hue='species')
plt.savefig('iris_pairplot.png')
sns.jointplot(x='sepal_length', y='sepal_width', data=iris, kind='scatter')
plt.savefig('iris_jointplot.png')
```

## Real Life Mini Project 2: Visualizing Customer Data Using Pair and Joint Plots

Project Overview:

In this project, we will use a sample dataset containing customer data, including age, income, and spending score. We'll explore the relationships between these variables to understand customer behavior using Pair and Joint plots.

Tasks:

### 1. Load the Customer Dataset

Load the customer dataset and inspect its structure.

### 2. Create a Pair Plot for All Variables

Use `sns.pairplot()` to create a grid of scatter plots showing the relationships between age, income, and spending score.

### 3. Customize Pair Plot with Hue

Add a hue parameter to distinguish customers by their spending category (e.g., High, Medium, Low).

### 4. Change Plot Type in Pair Plot

Change the type of the plot to display kernel density estimation (kind='kde').

### **5. Create a Joint Plot between Age and Spending Score**

Use sns.jointplot() to visualize the relationship between age and spending score using scatter and histograms.

### **6. Use a Hexagonal Plot in Joint Plot**

Replace the scatter plot with hexagonal binning by setting kind='hex'.

### **7. Add Kernel Density Estimation (KDE) in Joint Plot**

Change the joint plot to a KDE plot to visualize the bivariate distribution.

### **8. Add a Regression Line in Joint Plot**

Use kind='reg' to add a regression line to the joint plot for better trend analysis.

### **9. Customizing Colors and Themes**

Customize the colors of the plots using the palette and color parameters.

### **10. Explore Relationships Between Income and Spending Score**

Create another pair plot and joint plot focusing on income and spending score.

### **11. Handle Missing Data**

Preprocess the dataset to handle missing values before visualizing the plots.

### **12. Visualize Customer Segments**

Create a pair plot to compare customer segments based on age, income, and spending score.

### **13. Export and Save the Plots**

Save the pair plot and joint plot as PNG images.

#### Step-by-Step Implementation:

```
import seaborn as sns
```

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

```
# Step 1: Load the Customer Dataset (Replace with your dataset)
```

```
# For illustration, we'll use the `sns.load_dataset('tips')` dataset
```

```
# In a real case, replace it with `sns.load_dataset('your_dataset')` or load from a CSV.
```

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

```
# Step 2: Create a Pair Plot for All Variables
```

```
sns.pairplot(customer_data, hue='sex', palette='coolwarm')
```

```
plt.show()
```

```
# Step 3: Customize Pair Plot with Hue
```

```
sns.pairplot(customer_data, hue='sex', palette='coolwarm', kind='scatter')
```

```
plt.show()
```

```
# Step 4: Change Plot Type in Pair Plot (using KDE)
```

```
sns.pairplot(customer_data, hue='sex', kind='kde')
```

```
plt.show()
```

```
# Step 5: Create a Joint Plot between Age and Spending Score (replace with actual columns)
```

```
sns.jointplot(x='total_bill', y='tip', data=customer_data, kind='scatter')
```

```
plt.show()
```

```
# Step 6: Use a Hexagonal Plot in Joint Plot
sns.jointplot(x='total_bill', y='tip', data=customer_data, kind='hex', color='blue')
plt.show()

# Step 7: Add Kernel Density Estimation (KDE) in Joint Plot
sns.jointplot(x='total_bill', y='tip', data=customer_data, kind='kde',
color='orange')
plt.show()

# Step 8: Add a Regression Line in Joint Plot
sns.jointplot(x='total_bill', y='tip', data=customer_data, kind='reg', color='green')
plt.show()

# Step 9: Customizing Colors and Themes
sns.pairplot(customer_data, hue='sex', palette='dark')
plt.show()

# Step 10: Explore Relationships Between Income and Spending Score
sns.jointplot(x='total_bill', y='tip', data=customer_data, kind='scatter')
plt.show()

# Step 11: Handle Missing Data (if any)
customer_data = customer_data.dropna()

# Step 12: Visualize Customer Segments
sns.pairplot(customer_data, hue='sex', palette='Set1')
plt.show()

# Step 13: Export and Save the Plots
sns.pairplot(customer_data, hue='sex')
```

```
plt.savefig('customer_pairplot.png')
sns.jointplot(x='total_bill', y='tip', data=customer_data, kind='scatter')
plt.savefig('customer_jointplot.png')
```

## Summary:

In both projects, we used **Pair Plots** to explore relationships between multiple features and **Joint Plots** to examine relationships between two variables with additional visual elements (like histograms or regression lines). These visualizations provide insights into the dataset, helping to identify correlations, trends, and anomalies that can guide further analysis or decision-making.

## Real Life Mini Project: Analyzing the Wine Quality Dataset Using Pair and Joint Plots

### Project Overview:

In this project, we will analyze the **Wine Quality Dataset** to explore relationships between different features (e.g., alcohol content, pH level, sulfur dioxide) and how they relate to the wine quality. We'll use **Pair Plots** to visualize pairwise relationships between various chemical properties and **Joint Plots** to focus on the relationship between two key features of the wine, such as alcohol content and quality.

### Tasks:

#### 1. Load the Wine Quality Dataset

2. Load the wine quality dataset using Seaborn's sns.load\_dataset() or from a CSV file. Understand the structure of the dataset.

### **3. Create a Pair Plot for All Variables**

Create a pair plot that visualizes pairwise relationships between multiple features, such as alcohol content, volatile acidity, citric acid, residual sugar, pH, and quality.

### **4. Customize Pair Plot with Hue for Quality**

Customize the pair plot to differentiate wines by their quality using the hue parameter.

### **5. Change Plot Kind in Pair Plot**

Change the kind parameter in the pair plot to show kernel density estimates (kind='kde') rather than scatter plots.

### **6. Adjust Plot Size**

Modify the height parameter of the pair plot to change the size of the plots.

### **7. Create a Joint Plot between Alcohol and Quality**

Use sns.jointplot() to visualize the relationship between alcohol content and wine quality with scatter plots and histograms.

### **8. Change the Joint Plot to Hex Plot**

Change the joint plot to a hexagonal plot to better visualize the density of points between alcohol content and quality.

### **9. Use KDE in the Joint Plot**

Use the kde kind in the joint plot to visualize the smooth distribution of the data between alcohol content and quality.

### **10.Add Regression Line to Joint Plot**

Add a regression line to the joint plot to observe the relationship between alcohol content and wine quality.

### **11.Visualize the Relationship Between pH and Quality**

Create another joint plot between pH and wine quality using scatter plots and histograms.

### **12.Explore the Relationship Between Residual Sugar and Quality**

Create a pair plot to investigate how residual sugar correlates with other features like alcohol and quality.

### **13.Handle Missing Data**

Clean the dataset by handling any missing or null values before creating the plots.

### **14.Save and Export the Plots**

Save the created pair plot and joint plots as PNG images using plt.savefig() for further analysis or presentation.

### **Step-by-Step Instructions:**

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

```
# Step 1: Load the Wine Quality Dataset
```

```
# If you don't have the dataset, you can use the following line:  
wine_data = sns.load_dataset('wine') # Replace with actual dataset path if  
loading from CSV  
  
# Step 2: Create a Pair Plot for All Variables  
sns.pairplot(wine_data, hue='quality', palette='viridis')  
plt.show()  
  
# Step 3: Customize Pair Plot with Hue for Quality  
sns.pairplot(wine_data, hue='quality', palette='coolwarm', kind='scatter')  
plt.show()  
  
# Step 4: Change Plot Kind in Pair Plot (using KDE)  
sns.pairplot(wine_data, hue='quality', kind='kde', height=2.5)  
plt.show()  
  
# Step 5: Adjust Plot Size  
sns.pairplot(wine_data, hue='quality', height=3)  
plt.show()  
  
# Step 6: Create a Joint Plot between Alcohol and Quality  
sns.jointplot(x='alcohol', y='quality', data=wine_data, kind='scatter')  
plt.show()  
  
# Step 7: Change the Joint Plot to Hex Plot  
sns.jointplot(x='alcohol', y='quality', data=wine_data, kind='hex', color='green')  
plt.show()  
  
# Step 8: Use KDE in the Joint Plot  
sns.jointplot(x='alcohol', y='quality', data=wine_data, kind='kde', color='orange')  
plt.show()
```

```
# Step 9: Add Regression Line to Joint Plot
sns.jointplot(x='alcohol', y='quality', data=wine_data, kind='reg', color='purple')
plt.show()

# Step 10: Visualize the Relationship Between pH and Quality
sns.jointplot(x='pH', y='quality', data=wine_data, kind='scatter')
plt.show()

# Step 11: Explore the Relationship Between Residual Sugar and Quality
sns.pairplot(wine_data, hue='quality', vars=['residual_sugar', 'quality', 'alcohol'])
plt.show()

# Step 12: Handle Missing Data
wine_data = wine_data.dropna() # Dropping rows with missing values (optional)

# Step 13: Save and Export the Plots
sns.pairplot(wine_data, hue='quality', palette='coolwarm')
plt.savefig('wine_pairplot.png')

sns.jointplot(x='alcohol', y='quality', data=wine_data, kind='scatter')
plt.savefig('wine_jointplot.png')
```

## Summary:

In this project, you will:

- **Use Pair Plots** to analyze relationships between multiple features of wine and how they relate to the wine quality.

- **Customize Pair Plots** to differentiate wines based on quality.
- **Use Joint Plots** to focus on two key features (like alcohol content and quality) and experiment with different plot types (scatter, hex, kde, reg).
- **Save the plots** for further use or reporting.

This project will help in understanding how different wine characteristics interact with each other and their impact on quality, providing valuable insights for wine producers and enthusiasts.

## Real Life Mini Project 1: Analyzing Customer Data with Pair and Joint Plots

### Project Overview:

In this project, we will analyze customer data (e.g., age, income, spending score, etc.) from a retail company. We will use **Pair Plots** to visualize pairwise relationships between various customer features and **Joint Plots** to understand the relationship between spending scores and income levels, helping the company tailor their marketing strategy.

### Tasks:

1. **Load the Customer Data**
2. Load a customer data CSV file containing information like age, income, spending score, etc.
3. **Explore the Data Structure**

Understand the dataset, check for missing values, and identify key features like age, income, spending score, etc.

#### **4. Create a Pair Plot for All Variables**

Use sns.pairplot() to visualize pairwise relationships between customer age, income, and spending score.

#### **5. Use hue to Differentiate Customer Segments**

Use the hue parameter to separate the data by customer segment or region (if available).

#### **6. Change Plot Kind in Pair Plot to KDE**

Change the plot kind to KDE (kernel density estimation) to visualize the smooth distribution of the data.

#### **7. Create a Joint Plot for Age and Income**

Use sns.jointplot() to visualize the relationship between customer age and income with scatter plots and histograms.

#### **8. Visualize the Joint Plot Using a Hex Plot**

Change the joint plot to a hex plot to visualize the density of points for age and income.

#### **9. Use KDE in the Joint Plot for Age and Income**

Use the kde kind in the joint plot to visualize a smoother distribution between age and income.

#### **10. Add a Regression Line in the Joint Plot**

Add a regression line to the joint plot for better insight into the correlation between age and income.

#### **11. Visualize the Relationship Between Spending Score and Income**

Create a joint plot between spending score and income to see how they are related.

### **12. Analyze Spending Score Across Different Age Groups**

Create a pair plot to explore the spending score distribution across different age groups.

### **13. Handle Missing Data (if any)**

Clean the dataset by handling any missing values through imputation or removal.

### **14. Save and Export the Pair and Joint Plots**

Save the final pair plot and joint plots as PNG images for future analysis or reports.

## **Real Life Mini Project 2: Visualizing Iris Flower Dataset with Pair and Joint Plots**

Project Overview:

In this project, we will use the **Iris Flower Dataset**, a popular dataset in machine learning, to analyze the relationships between different flower attributes (sepal length, sepal width, petal length, and petal width). We will use **Pair Plots** to visualize relationships between multiple attributes and **Joint Plots** to focus on key relationships, such as petal length and petal width.

Tasks:

### **1. Load the Iris Flower Dataset**

Load the Iris flower dataset from Seaborn's built-in dataset or from a CSV file.

## 2. Explore the Dataset

Understand the dataset by checking its structure and the feature columns such as sepal length, sepal width, petal length, and petal width.

## 3. Create a Pair Plot to Visualize Relationships

Use sns.pairplot() to visualize the pairwise relationships between sepal length, sepal width, petal length, and petal width.

## 4. Add Color Differentiation Using hue

Use the hue parameter to color the data points based on the flower species (Setosa, Versicolor, Virginica).

## 5. Change the Plot Kind in the Pair Plot

Modify the pair plot to use KDE plots instead of scatter plots to show the smooth distribution of data.

## 6. Create a Joint Plot for Petal Length and Petal Width

Use sns.jointplot() to visualize the relationship between petal length and petal width with scatter plots and histograms.

## 7. Use a Hex Plot for Joint Plot

Change the joint plot to a hex plot to visualize the density of points between petal length and petal width.

## 8. Use KDE in the Joint Plot for Petal Length and Petal Width

Use the kde plot kind in the joint plot to visualize the distribution of petal length and petal width.

## 9. Add a Regression Line to the Joint Plot

Add a regression line to the joint plot to understand the linear relationship between petal length and petal width.

### **10. Explore the Relationship Between Sepal Width and Sepal Length**

Create a joint plot to explore the relationship between sepal width and sepal length.

### **11. Visualize Petal Length vs Sepal Length**

Create another pair plot or joint plot to explore the relationship between petal length and sepal length.

### **12. Handle Missing Data**

If there is any missing data, handle it by filling or removing the missing entries.

### **13. Export the Plots**

Save the generated pair plots and joint plots as PNG or PDF files for reporting purposes.

These two projects focus on practical uses of pair plots and joint plots in data visualization, with an emphasis on exploring relationships between different features and understanding the impact of categories like customer segments or flower species.

# Day 66

## Regression & Time Series Plots

What is Regression Analysis?

**Regression Analysis** is a statistical technique used to understand the relationship between variables. The primary purpose of regression is to model the relationship between a dependent variable (also known as the target variable) and one or more independent variables (predictors).

In the context of data visualization:

- **Linear Regression** represents a relationship where the dependent variable changes linearly with the independent variable.
- **Regression Analysis** can be used for prediction, trend analysis, and understanding how one variable impacts another.

Creating a Regression Plot (`sns.regplot()`)

A **Regression Plot** is a type of plot used to show the relationship between two variables, specifically to see if there's a linear trend. The `sns.regplot()` function in Seaborn fits and visualizes a regression line along with the scatter plot.

Syntax for `sns.regplot()`

```
sns.regplot(x, y, data=None, scatter=True, line_kws=None, scatter_kws=None)
```

- **x**: The independent variable (predictor).

- **y**: The dependent variable (target).
- **data**: The DataFrame containing the data.
- **scatter**: Boolean (default: True), whether to plot the scatter points.
- **line\_kws**: Dictionary of keyword arguments for customizing the regression line (e.g., color, linewidth).
- **scatter\_kws**: Dictionary of keyword arguments for customizing scatter points (e.g., color, size).

#### Example: Simple Linear Regression Plot

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

# Sample Data
data = {
    'Hours_Studied': [1, 2, 3, 4, 5, 6, 7, 8],
    'Scores': [25, 45, 55, 60, 70, 80, 85, 95]
}

df = pd.DataFrame(data)

# Create a regression plot
sns.regplot(x='Hours_Studied', y='Scores', data=df)

# Customize the title and show the plot
plt.title('Regression Plot: Hours Studied vs Scores')
plt.show()
```

### Explanation:

- This plot shows the relationship between the number of hours studied (Hours\_Studied) and the scores (Scores).
- The line shows the predicted relationship between hours studied and the corresponding score.

### Customizing Regression Lines

You can customize the regression line using the line\_kws parameter to change its appearance, such as its color or line style.

```
sns.regplot(x='Hours_Studied', y='Scores', data=df, line_kws={'color': 'red',  
'linewidth': 2})  
plt.title('Customized Regression Line')  
plt.show()
```

In this case, the regression line is customized to be red and have a width of 2.

### Creating Residual Plots (sns.residplot())

A **Residual Plot** shows the difference between the actual data points and the regression line. It is useful to check how well the regression model fits the data. A good model should have randomly scattered residuals.

### Syntax for sns.residplot()

```
sns.residplot(x, y, data=None, lowess=False, line_kws=None, scatter_kws=None)
```

- **x:** The independent variable.
- **y:** The dependent variable.
- **data:** The DataFrame containing the data.
- **lowess:** If True, it fits a locally weighted scatterplot smoothing curve.
- **line\_kws:** Dictionary of keyword arguments for customizing the line (e.g., color, linewidth).
- **scatter\_kws:** Dictionary of keyword arguments for customizing the scatter points.

### Example: Residual Plot

```
# Create a residual plot
sns.residplot(x='Hours_Studied', y='Scores', data=df, lowess=True,
               scatter_kws={'color': 'blue'})
plt.title('Residual Plot: Hours Studied vs Scores')
plt.show()
```

### Explanation:

- The residual plot visualizes the difference between the actual data points and the regression line.
- You can use lowess=True to add a smoothing curve to help identify patterns.

## Time Series Visualization using Seaborn

**Time Series** plots are used to visualize data that is collected over time (e.g., stock prices, temperature). Time Series plots help to identify trends, seasonal patterns, and outliers over a period.

Seaborn provides a simple way to plot time series data with `sns.lineplot()`.

### Syntax for `sns.lineplot()`

```
sns.lineplot(x=None, y=None, data=None, hue=None, style=None, markers=None)
```

- **x:** The time variable.
- **y:** The dependent variable (value you want to analyze over time).
- **data:** The DataFrame containing the time series data.
- **hue:** Used for color grouping.
- **style:** Different line styles for grouping.
- **markers:** To mark the points on the line plot.

### Example: Time Series Plot

```
import seaborn as sns
import pandas as pd

# Sample time series data
data = {
    'Date': pd.date_range(start='2021-01-01', periods=10, freq='D'),
    'Sales': [150, 200, 220, 240, 300, 350, 400, 450, 500, 550]
}
```

```
df = pd.DataFrame(data)

# Plot the time series data
sns.lineplot(x='Date', y='Sales', data=df)
plt.title('Time Series Plot: Sales Over Time')
plt.xlabel('Date')
plt.ylabel('Sales')
plt.xticks(rotation=45) # Rotate x-axis labels for better readability
plt.show()
```

### Explanation:

- This plot visualizes the sales data over a 10-day period.
- The x axis represents the date, and the y axis represents the sales for that particular date.

### Summary

- **Regression Analysis** helps identify relationships between variables.
- **sns.regplot()** is used to create regression plots and visualize these relationships.
- **Residual plots** help evaluate how well a regression model fits the data by checking the errors between actual and predicted values.
- **Time Series Plots** are used to visualize data over time, often used for trend analysis.

These techniques are crucial for predictive modeling, understanding data relationships, and analyzing time-based patterns.

Here are two real-life mini-project requirements with step-by-step implementation for **Regression & Time Series Plots**:

### **Mini Project 1: Predicting House Prices Using Regression Analysis**

**Objective:** Build a regression model to predict house prices based on various features such as square footage, number of rooms, etc. Then, visualize the relationship between square footage and house price using regression plots.

Tasks:

1. **Load Data:** Import a housing dataset that includes features such as square footage, number of rooms, and house prices.
2. **Data Preprocessing:** Clean the data by handling missing values and ensure that the data is in the correct format for regression.
3. **Exploratory Data Analysis (EDA):** Visualize the relationship between square footage and house price.
4. **Create a Regression Plot:** Use sns.regplot() to visualize the linear relationship between square footage and house price.
5. **Customize Regression Lines:** Change the color and style of the regression line.
6. **Residual Plot:** Create a residual plot using sns.residplot() to check the accuracy of the regression model.
7. **Modeling:** Use linear regression from sklearn to fit the data and create a prediction model.
8. **Predictions:** Use the fitted model to make predictions for house prices based on square footage.

9. **Evaluate the Model:** Evaluate the model using R-squared and mean squared error.
10. **Time Series:** Create a time series plot to visualize how house prices have changed over time (if applicable).

## Step-by-Step Implementation:

### 1. Importing Required Libraries

```
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
```

### 2. Loading the Dataset

```
# Load a sample dataset (for example, housing data)
data = sns.load_dataset("penguins") # Example dataset; replace with your
housing dataset
data.head()
```

### 3. Data Preprocessing

```
# Clean the dataset by removing rows with missing values
data = data.dropna(subset=['bill_length_mm', 'bill_depth_mm'])

# Ensure the relevant columns are numeric
```

```
data['bill_length_mm'] = pd.to_numeric(data['bill_length_mm'], errors='coerce')
```

#### 4. Create Regression Plot

```
# Visualizing the relationship between two variables (e.g., 'bill_length_mm' and  
'bill_depth_mm')  
sns.regplot(x='bill_length_mm', y='bill_depth_mm', data=data)  
plt.title('Regression Plot: Bill Length vs Bill Depth')  
plt.show()
```

#### 5. Customizing the Regression Line

```
sns.regplot(x='bill_length_mm', y='bill_depth_mm', data=data, line_kws={'color':  
'red', 'linewidth': 2})  
plt.title('Customized Regression Plot')  
plt.show()
```

#### 6. Creating a Residual Plot

```
sns.residplot(x='bill_length_mm', y='bill_depth_mm', data=data, lowess=True,  
scatter_kws={'color': 'blue'})  
plt.title('Residual Plot')  
plt.show()
```

#### 7. Fit the Regression Model

```
# Assuming 'bill_length_mm' is the independent variable and 'bill_depth_mm' is  
the dependent variable
```

```
X = data[['bill_length_mm']] # Feature
y = data['bill_depth_mm'] # Target
```

```
# Fit the model
model = LinearRegression()
model.fit(X, y)
```

```
# Make predictions
y_pred = model.predict(X)
```

## 8. Model Evaluation

```
# Evaluate the model
mse = mean_squared_error(y, y_pred)
r2 = r2_score(y, y_pred)
print(f'Mean Squared Error: {mse}')
print(f'R-squared: {r2}')
```

## Mini Project 2: Visualizing Stock Prices Over Time

**Objective:** Visualize stock price trends over time using time series plots and perform regression analysis to model future prices based on historical data.

Tasks:

1. **Load Stock Data:** Import stock price data (e.g., using yfinance or pandas\_datareader).

2. **Preprocess Data:** Ensure that the data is in the correct format (date format) and handle any missing values.
3. **Time Series Plot:** Create a time series plot to visualize how stock prices have evolved over time.
4. **Regression Analysis:** Perform regression to predict future stock prices based on historical data.
5. **Regression Plot:** Use sns.regplot() to visualize the relationship between historical and predicted stock prices.
6. **Customize Regression Line:** Change the regression line's style for better visualization.
7. **Residual Plot:** Check for any patterns or trends in the residual plot.
8. **Predict Future Prices:** Use the model to predict future stock prices.
9. **Model Evaluation:** Evaluate the performance of the model using metrics like R-squared.
10. **Time Series Forecasting:** Create a forecast for future stock prices and visualize it.

### **Step-by-Step Implementation:**

1. Install yfinance and Import Libraries

```
pip install yfinance  
import yfinance as yf  
import seaborn as sns  
import matplotlib.pyplot as plt  
import pandas as pd  
from sklearn.linear_model import LinearRegression
```

## 2. Load Stock Data

```
# Download stock price data (e.g., for Tesla)
data = yf.download('TSLA', start='2015-01-01', end='2020-12-31')
data.head()
```

## 3. Preprocess Data

```
# Check for missing values
data.isnull().sum()

# Drop rows with missing values (if any)
data = data.dropna(subset=['Close'])

# Reset index to use the Date as the index
data = data.reset_index()
```

## 4. Create a Time Series Plot

```
# Plotting the closing stock prices over time
sns.lineplot(x='Date', y='Close', data=data)
plt.title('Stock Price Over Time')
plt.xlabel('Date')
plt.ylabel('Stock Price')
plt.xticks(rotation=45) # Rotate x-axis labels for readability
plt.show()
```

## 5. Perform Regression Analysis

```
# Create a linear regression model to predict future stock prices
X = pd.to_numeric(data['Date'].map(pd.Timestamp.timestamp)).values.reshape(-
1, 1) # Convert dates to numeric
y = data['Close'].values # Target: closing prices

model = LinearRegression()
model.fit(X, y)

# Predict stock prices
y_pred = model.predict(X)
```

## 6. Regression Plot

```
sns.regplot(x=pd.to_numeric(data['Date'].map(pd.Timestamp.timestamp)),
y='Close', data=data, line_kws={'color': 'red'})
plt.title('Stock Price Regression')
plt.show()
```

## 7. Customize Regression Line

```
sns.regplot(x=pd.to_numeric(data['Date'].map(pd.Timestamp.timestamp)),
y='Close', data=data,
            line_kws={'color': 'green', 'linewidth': 3})
plt.title('Customized Regression Line')
plt.show()
```

## 8. Residual Plot

```
sns.residplot(x=pd.to_numeric(data['Date'].map(pd.Timestamp.timestamp)),  
y='Close', data=data, lowess=True)  
plt.title('Residual Plot for Stock Price Regression')  
plt.show()
```

## 9. Predict Future Prices

```
# Create a new set of dates for prediction (for the next 30 days)  
future_dates = pd.date_range(start='2021-01-01', periods=30, freq='D')  
  
# Convert these dates to numeric timestamps  
future_dates_numeric =  
pd.to_numeric(future_dates.map(pd.Timestamp.timestamp)).values.reshape(-1,  
1)  
  
# Make predictions  
future_predictions = model.predict(future_dates_numeric)  
  
# Display predictions  
predicted_stock_prices = pd.DataFrame({'Date': future_dates, 'Predicted Price':  
future_predictions})  
print(predicted_stock_prices)
```

## 10. Model Evaluation

```
# Evaluate the model's performance (R-squared)  
r2 = model.score(X, y)
```

```
print(f'R-squared: {r2}')
```

## Mini Project: Predicting Monthly Sales Using Regression and Time Series Analysis

**Objective:** Build a regression model to predict monthly sales for a retail store based on historical data, then visualize the results using regression plots and time series visualization.

Tasks:

1. **Load and Explore the Data:** Import a dataset containing monthly sales data, including sales numbers and dates.
2. **Data Preprocessing:** Clean the data by handling any missing values and ensuring proper data types for time and sales columns.
3. **Visualize Sales Over Time:** Create a simple line plot to visualize the sales data over time (monthly sales vs. date).
4. **Create a Regression Plot:** Use sns.regplot() to visualize the relationship between time (in months) and sales.
5. **Customize Regression Line:** Modify the appearance of the regression line (e.g., change the line color and style) to enhance readability.
6. **Create Residual Plot:** Create a residual plot using sns.residplot() to check the accuracy of the regression model.
7. **Fit the Regression Model:** Fit a linear regression model to the sales data to predict sales based on time.
8. **Evaluate the Regression Model:** Evaluate the performance of the regression model using metrics like R-squared and mean squared error.

9. **Predict Future Sales:** Use the regression model to predict future sales for upcoming months.
10. **Visualize Sales Predictions:** Create a time series plot that includes both historical sales and predicted sales.
11. **Check for Seasonality:** Perform an analysis to check for any seasonal patterns in the sales data using time series decomposition.
12. **Forecast Future Sales:** Use the regression model and/or time series analysis to forecast the next 6 months of sales.
13. **Time Series Visualization:** Create a time series visualization of the sales data with annotations to highlight any trends, seasonal variations, or anomalies.

## Mini Project 1: Predicting House Prices Based on Features

**Objective:** Use regression analysis to predict house prices based on features such as size, number of rooms, and location. Visualize the relationships and residuals, and analyze time-series trends in house prices.

Tasks:

1. **Load and Inspect the Dataset:** Import a dataset with features like house size, number of rooms, location, and prices.
2. **Data Preprocessing:** Clean the data by filling missing values, converting categorical data into numerical form (if any), and scaling the features.
3. **Explore the Relationship Between Size and Price:** Create a scatter plot to visualize the relationship between the size of the house and its price.
4. **Create a Regression Plot:** Use sns.regplot() to visualize the regression line between house size and price.

5. **Customizing the Regression Line:** Customize the regression plot by changing the color and style of the regression line to make the plot clearer.
6. **Check for Outliers:** Create a residual plot using `sns.residplot()` to visualize outliers and the fit of the regression model.
7. **Fit the Regression Model:** Fit a linear regression model to predict the house price based on size and other features.
8. **Evaluate the Model:** Calculate the R-squared value and mean squared error of the regression model to assess its accuracy.
9. **Predict House Prices:** Predict the price of houses based on the model for various house sizes.
10. **Visualize the Predictions:** Create a plot showing both actual house prices and predicted prices for comparison.
11. **Time Series Analysis of Prices:** If the dataset includes time-related data (e.g., sale dates), create a time series plot to analyze how prices have changed over time.
12. **Create Seasonal Decomposition:** Use time series decomposition to analyze seasonal trends in the housing market.
13. **Forecast Future Prices:** Using regression and time series analysis, forecast future house prices for the next quarter.

## Mini Project 2: Analyzing Stock Market Data and Forecasting Future Prices

**Objective:** Analyze historical stock market data and forecast future stock prices using regression analysis and time series visualization.

Tasks:

1. **Load the Stock Market Data:** Import stock market data containing features like stock prices, date, and trading volume.
2. **Data Preprocessing:** Clean the dataset by filling missing values, converting dates to datetime format, and normalizing the price values if necessary.
3. **Visualize Stock Prices Over Time:** Create a line plot to visualize how the stock prices have fluctuated over time.
4. **Apply Regression Analysis:** Use sns.regplot() to visualize the relationship between stock prices and trading volume.
5. **Customize the Regression Plot:** Customize the regression plot by modifying plot attributes like color, markers, and line styles.
6. **Residual Plot:** Create a residual plot using sns.residplot() to evaluate the fit of the regression model and check for any patterns in the residuals.
7. **Time Series Visualization:** Create a time series plot to show how the stock price has changed day by day.
8. **Fit a Linear Regression Model:** Fit a regression model to predict the future stock price based on historical data.
9. **Evaluate the Model:** Calculate the R-squared value, mean squared error, and other evaluation metrics to assess the regression model's performance.
10. **Forecast Stock Prices:** Use the regression model to forecast stock prices for the next few days or weeks.
11. **Create Moving Averages:** Calculate and visualize moving averages of stock prices to identify trends.
12. **Check for Seasonal Patterns:** If the stock market data spans over a long period, perform seasonal decomposition to check for seasonal patterns in the stock prices.
13. **Future Price Prediction:** Using regression and time series analysis, predict the stock price for the next month and visualize the forecasted trend.

These two mini projects allow you to dive into regression and time series analysis with practical, real-world datasets. You'll learn to predict outcomes, evaluate models, and handle time-dependent data.

## Day 67

### Advanced Seaborn Customization & Styling

Seaborn is a powerful Python library for statistical data visualization, built on top of Matplotlib. It provides an easy-to-use interface for generating a variety of plots. Seaborn also allows for extensive customization and styling, enabling you to create visually appealing plots with minimal code.

Below, we will cover **Advanced Seaborn Customization & Styling**, including changing themes, customizing figure sizes, creating multi-panel plots, and saving/exporting visualizations.

#### 1. Changing Seaborn Themes (`sns.set_theme()`)

**Definition:** The `sns.set_theme()` function in Seaborn allows you to set the style and color palette of your plots globally. This can make your plots more visually appealing by giving them a consistent theme. You can set the background color, gridlines, and color palette all at once.

Syntax:

```
sns.set_theme(style="darkgrid", palette="deep", font="sans-serif", rc=None)
```

- **style**: Sets the background style of the plot (e.g., 'white', 'darkgrid', 'whitegrid', 'ticks').
- **palette**: Controls the color scheme used in the plot (e.g., 'deep', 'muted', 'dark', etc.).
- **font**: Specifies the font used for text (e.g., 'sans-serif', 'serif').
- **rc**: A dictionary of parameters used to control more advanced aspects of the plot's appearance.

Example:

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

# Set a darkgrid theme with a deep color palette
sns.set_theme(style="darkgrid", palette="deep")

# Generate some random data
data = np.random.randn(100)

# Create a simple Seaborn plot
sns.histplot(data)

plt.show()
```

## 2. Customizing Figure Sizes, Grid Styles, and Color Palettes

**Definition:** You can customize figure sizes, grid styles, and color palettes in Seaborn to control the appearance of the plots. This can be useful when presenting plots in different contexts, such as reports or presentations.

### Customizing Figure Size:

You can adjust the size of the figure by using `matplotlib.pyplot.figure()` before creating the plot.

#### Syntax:

```
plt.figure(figsize=(width, height))
```

#### Example:

```
# Set the figure size to (10, 6)
plt.figure(figsize=(10, 6))
```

```
# Create the plot
sns.histplot(data)
plt.show()
```

### Grid Styles:

You can control the gridlines by adjusting the Seaborn theme.

```
sns.set_theme(style="whitegrid") # Gridlines are shown on a white background
```

### Color Palettes:

Seaborn provides various predefined color palettes. You can specify a palette to control the color scheme.

```
sns.set_palette("muted") # Set muted color palette
```

Example:

```
sns.set_theme(style="whitegrid", palette="muted")
plt.figure(figsize=(10, 6))
sns.histplot(data)
plt.show()
```

### 3. Combining Multiple Plots using FacetGrid (sns.FacetGrid())

**Definition:** sns.FacetGrid() is used to create grids of subplots based on a categorical variable, allowing you to compare multiple aspects of a dataset across different subsets.

Syntax:

```
g = sns.FacetGrid(data, col="category_column", row="row_column",
hue="hue_column")
g.map(plot_function, x_column, y_column)
g.add_legend()
```

- **data**: The data you want to plot.
- **col**: The categorical variable that determines the columns.
- **row**: The categorical variable that determines the rows.
- **hue**: The variable that controls color coding.

Example:

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

# Load the example dataset
tips = sns.load_dataset("tips")

# Create a FacetGrid to plot total bill against tip, split by day
g = sns.FacetGrid(tips, col="day")
g.map(sns.scatterplot, "total_bill", "tip")
g.add_legend()

plt.show()
```

#### 4. Creating Multi-Panel Plots using sns.catplot()

**Definition:** sns.catplot() combines multiple categorical plots (like bar, box, violin, etc.) into a single grid. It's useful for comparing multiple categories at once.

Syntax:

```
sns.catplot(data=data, kind="plot_type", x="x_column", y="y_column",
hue="hue_column")
```

- **data:** The dataset.
- **kind:** The type of categorical plot ('bar', 'box', 'violin', 'swarm', etc.).
- **x:** The categorical variable for the x-axis.
- **y:** The continuous variable for the y-axis.
- **hue:** Optional, used for color-coding the data.

Example:

```
# Create a multi-panel plot with bar plots for each day in the dataset  
sns.catplot(x="day", y="total_bill", kind="bar", data=tips)  
plt.show()
```

## 5. Saving & Exporting Seaborn Visualizations

**Definition:** Once you've created your visualizations, you can save and export them in various formats (e.g., PNG, JPEG, SVG) for use in reports or presentations.

Syntax:

```
plt.savefig("file_name.extension", dpi=300, bbox_inches="tight")
```

- **file\_name.extension:** The name and format of the output file (e.g., 'plot.png', 'plot.pdf').
- **dpi:** The resolution of the image (higher DPI means better quality).
- **bbox\_inches:** Controls the bounding box around the plot (use "tight" to avoid unnecessary space around the plot).

Example:

```
# Create a plot  
sns.histplot(data)  
  
# Save the plot as a PNG image with high resolution  
plt.savefig("seaborn_histogram.png", dpi=300, bbox_inches="tight")  
plt.show()
```

## Real-Life Example: Customizing and Saving a Seaborn Plot

Let's put all of the above customizations into practice. We'll create a customized multi-panel plot of the "tips" dataset and save the result.

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

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

# Set the Seaborn theme and palette
sns.set_theme(style="whitegrid", palette="coolwarm")

# Create a FacetGrid of bar plots for the total bill by day
g = sns.FacetGrid(tips, col="day")
g.map(sns.barplot, "time", "total_bill", estimator=sum)
g.set_axis_labels("Time of Day", "Total Bill ($)")
g.set_titles("{col_name} Day")
g.add_legend()

# Save the figure as a PNG image
plt.savefig("seaborn_tips_plot.png", dpi=300, bbox_inches="tight")

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

## Summary of Advanced Seaborn Customization & Styling:

- **Changing Themes:** You can set themes to adjust the overall look of your plots.
- **Customizing Appearance:** Customize plot sizes, grid styles, and color palettes.
- **FacetGrid:** Use FacetGrid to create multi-plot grids for comparing subsets of data.
- **catplot():** Create multiple categorical plots in one frame to visualize distributions across categories.
- **Saving Plots:** Save and export your visualizations in different formats for use in presentations, reports, etc.

This gives you control over the appearance and output of Seaborn plots, making it easier to share or present your findings in a more professional and visually appealing manner.

## Mini Project 1: E-Commerce Data Analysis with Seaborn Customization

In this project, you will analyze an e-commerce dataset and create visualizations using advanced Seaborn customization and styling. The dataset includes product sales data, with columns like category, price, quantity\_sold, day\_of\_week, and customer\_type. Your task is to customize the plots to make them visually appealing and informative.

Step-by-Step Implementation:

### Step 1: Import Libraries

973

Private & Confidential : Vetri Technology Solutions

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

# Load sample dataset (simulated e-commerce data)
data = pd.DataFrame({
    'category': np.random.choice(['Electronics', 'Furniture', 'Clothing'], size=200),
    'price': np.random.uniform(50, 500, 200),
    'quantity_sold': np.random.randint(1, 20, 200),
    'day_of_week': np.random.choice(['Monday', 'Tuesday', 'Wednesday',
    'Thursday', 'Friday'], size=200),
    'customer_type': np.random.choice(['New', 'Returning'], size=200)
})
```

### Step 2: Set Seaborn Theme

```
# Set the Seaborn theme to 'darkgrid' with a 'coolwarm' palette
sns.set_theme(style="darkgrid", palette="coolwarm")
```

### Step 3: Customizing Figure Sizes and Plot

```
# Create a larger figure size
```

```
plt.figure(figsize=(12, 6))
```

```
# Create a bar plot showing quantity sold by product category
```

```
sns.barplot(x='category', y='quantity_sold', data=data)
```

```
# Set plot title and labels
```

```
plt.title('Quantity Sold by Category')
```

```
plt.xlabel('Product Category')
plt.ylabel('Quantity Sold')
```

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

#### **Step 4: Using FacetGrid for Multiple Plots**

```
# Create a FacetGrid to show price distribution across different categories
g = sns.FacetGrid(data, col="category")
g.map(sns.histplot, "price", bins=10, kde=True)
g.set_axis_labels("Price", "Frequency")
g.set_titles("{col_name} Category")
```

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

#### **Step 5: Using catplot() for Multi-Panel Plot**

```
# Create a multi-panel plot using sns.catplot
sns.catplot(x="day_of_week", y="quantity_sold", hue="customer_type",
kind="box", data=data)
```

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

#### **Step 6: Save the Final Visualization**

```
# Save the last plot to a PNG file
sns.catplot(x="day_of_week", y="quantity_sold", hue="customer_type",
```

```
kind="box", data=data)
plt.savefig("ecommerce_sales_boxplot.png", dpi=300, bbox_inches="tight")
```

## Mini Project 2: Employee Performance Data Visualization

In this project, you will work with employee performance data and create visualizations to explore trends, relationships, and comparisons. The dataset includes columns like department, performance\_score, work\_hours, age, and experience\_level. You will use advanced Seaborn techniques to visualize and customize the data.

Step-by-Step Implementation:

### Step 1: Import Libraries

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

```
# Create simulated employee performance data
data = pd.DataFrame({
    'department': np.random.choice(['HR', 'Engineering', 'Sales'], size=150),
    'performance_score': np.random.uniform(1, 10, 150),
    'work_hours': np.random.uniform(30, 50, 150),
    'age': np.random.randint(22, 60, 150),
    'experience_level': np.random.choice(['Junior', 'Mid', 'Senior'], size=150)
})
```

## Step 2: Set Seaborn Theme

```
# Set the Seaborn theme to 'whitegrid' with a 'pastel' palette  
sns.set_theme(style="whitegrid", palette="pastel")
```

## Step 3: Customizing Figure Size and Grid Style

```
# Set a figure size for the plots  
plt.figure(figsize=(10, 6))
```

```
# Create a scatter plot showing performance score vs. work hours  
sns.scatterplot(x='work_hours', y='performance_score', data=data,  
hue='experience_level', style='experience_level')
```

```
# Show the plot  
plt.title('Performance Score vs. Work Hours')  
plt.xlabel('Work Hours')  
plt.ylabel('Performance Score')  
plt.show()
```

## Step 4: Combining Multiple Plots Using FacetGrid

```
# Create a FacetGrid to visualize the relationship between work hours and  
performance score by department  
g = sns.FacetGrid(data, col="department")  
g.map(sns.scatterplot, "work_hours", "performance_score",  
hue="experience_level", palette="Set1")
```

```
# Add titles and labels  
g.set_axis_labels("Work Hours", "Performance Score")  
g.set_titles("{col_name} Department")
```

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

### Step 5: Creating Multi-Panel Plots using sns.catplot

```
# Create a box plot for performance scores across experience levels  
sns.catplot(x="experience_level", y="performance_score", kind="box",  
data=data)
```

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

### Step 6: Saving and Exporting Visualizations

```
# Save the multi-panel plot as a PNG file  
sns.catplot(x="experience_level", y="performance_score", kind="box",  
data=data)  
plt.savefig("employee_performance_boxplot.png", dpi=300, bbox_inches="tight")
```

### Summary of Steps:

For Mini Project 1 (E-Commerce Data Analysis):

1. Import necessary libraries and load the dataset.
2. Set a Seaborn theme and customize the plot size.
3. Create bar and histograms to show quantity sold by category and price distribution.

4. Use FacetGrid to create multi-panel plots for different categories.
5. Create multi-panel plots using sns.catplot to show sales performance across days and customer types.
6. Save the visualization as an image file.

For Mini Project 2 (Employee Performance Visualization):

1. Import libraries and create a simulated employee dataset.
2. Set a Seaborn theme and customize figure size.
3. Create a scatter plot to visualize performance score vs. work hours.
4. Use FacetGrid to visualize the performance score based on different departments.
5. Create multi-panel plots using sns.catplot to show performance scores across experience levels.
6. Save and export the final plots.

These projects demonstrate how to use advanced Seaborn customization techniques, including setting themes, creating multiple plot types, and exporting visualizations for use in real-world applications.

## **Mini Project: Analyzing Car Sales Data with Advanced Seaborn Customization**

In this project, you will work with a car sales dataset that contains information about car models, prices, sale dates, and customer demographics. You will use Seaborn to create various visualizations with customized themes, figure sizes, and multi-panel plots. The goal is to analyze trends in car sales across different categories (e.g., model, customer age, sale region) and generate exportable visualizations.

Task List:

**1. Import Libraries and Dataset**

- a. Import Seaborn, Matplotlib, Pandas, and NumPy.
- b. Load the car sales dataset (CSV or simulated data) into a Pandas DataFrame.

**2. Set a Custom Seaborn Theme**

- a. Use sns.set\_theme() to set a custom theme for the visualizations (e.g., darkgrid, white, etc.).
- b. Apply a color palette that complements the data (e.g., color\_palette("coolwarm")).

**3. Examine the Data**

- a. Display the first few rows of the dataset.
- b. Check for missing values and handle any missing data appropriately (fill or drop).

**4. Customize Figure Size for Bar Plots**

- a. Create a bar plot showing the number of car sales by car model.
- b. Set a custom figure size (e.g., figsize=(10, 6)).

**5. Create a Distribution Plot for Car Prices**

- a. Create a distribution plot (sns.histplot()) for car prices.
- b. Adjust the number of bins and overlay a kernel density estimate (KDE).

**6. Use FacetGrid for Region-wise Car Sales Analysis**

- a. Use sns.FacetGrid() to create separate plots for car sales by region.
- b. Display the number of car sales per region in a bar plot.

**7. Create a Scatter Plot for Car Price vs. Age of Customer**

- a. Use sns.scatterplot() to visualize the relationship between car price and customer age.
- b. Customize the plot with different markers based on customer type (e.g., new or returning).

**8. Create a Box Plot for Car Sales by Region and Customer Age**

- a. Create a box plot using sns.boxplot() to show car price distribution by customer age and region.
- b. Customize the grid and plot appearance.

## 9. Create a Multi-Panel Plot to Show Sales Trends Over Time

- a. Use sns.catplot() to create multi-panel plots showing car sales trends over time.
- b. Display sales trends per region with customized x and y axes.

## 10. Customize Color Palette for the Plots

- a. Choose and apply a custom color palette for all plots (e.g., sns.set\_palette("Set2")).
- b. Ensure that color schemes are consistent throughout all plots.

## 11. Add Annotations to the Plot

- a. Add text annotations to the scatter plot showing key points (e.g., highest and lowest sales).
- b. Customize the font size, color, and position of annotations.

## 12. Save the Plots to PNG and SVG Files

- a. Save the final visualizations to .png and .svg files using plt.savefig().
- b. Ensure the plot's resolution is set to a high value (e.g., dpi=300).

## 13. Create a Combined Plot (Heatmap of Correlations)

- a. Create a heatmap of correlations between numeric variables such as car price, sales volume, and customer age using sns.heatmap().
- b. Customize the heatmap's color map and add annotations for clarity.

Goal:

This mini project will help you practice using Seaborn's advanced customization features, including setting themes, customizing figure sizes, combining multiple plots with FacetGrid, and creating multi-panel visualizations. You'll also learn how to save your plots and export them in different formats for reporting or presentations.

## Mini Project 1: Analyzing E-commerce Data with Advanced Seaborn Customization

In this project, you will work with an e-commerce dataset containing information about products, categories, sales, and customer demographics. You will create customized visualizations to analyze sales trends, customer behavior, and product performance across different categories.

Task List:

### 1. Import Libraries and Dataset

- a. Import Seaborn, Matplotlib, Pandas, and NumPy.
- b. Load the e-commerce sales dataset into a Pandas DataFrame.

### 2. Set a Custom Seaborn Theme

- a. Use `sns.set_theme()` to set a theme that matches the aesthetic of the report (e.g., darkgrid or white).

### 3. Examine the Data

- a. Display the first few rows of the dataset to inspect the data.
- b. Handle any missing values by either filling them with appropriate values or removing rows with null values.

### 4. Sales Distribution by Product Category

- a. Create a bar plot showing sales by product category.
- b. Set the figure size to ensure readability.

### 5. Visualize Customer Age Distribution

- a. Create a histogram (`sns.histplot()`) to display the distribution of customer ages.
- b. Customize the number of bins and add a KDE curve.

### 6. Sales Trends Over Time (Line Plot)

- a. Use a line plot (`sns.lineplot()`) to show sales trends over time (e.g., daily or monthly).
- b. Add customization for line styles, markers, and color palettes.

## 7. FacetGrid: Sales by Region

- a. Use `sns.FacetGrid()` to create separate plots showing sales by region.
- b. Display the number of products sold in each region.

## 8. Product Pricing vs. Sales Volume (Scatter Plot)

- a. Create a scatter plot showing the relationship between product prices and sales volume.
- b. Use color encoding to differentiate between categories.

## 9. Box Plot: Sales Price by Product Category

- a. Create a box plot (`sns.boxplot()`) to visualize the sales price distribution for different product categories.
- b. Customize the grid style and figure size.

## 10. FacetGrid: Customer Gender vs. Sales

- a. Use `sns.FacetGrid()` to show the sales distribution based on customer gender.
- b. Include multiple subplots with different aspects of the data.

## 11. Save and Export All Plots

- a. Save the generated plots in both `.png` and `.svg` formats.
- b. Ensure that the resolution is set to a high value (e.g., `dpi=300`).

## 12. Multi-Panel Plot for Product Sales by Category and Region

- a. Create a multi-panel plot using `sns.catplot()` to show the sales of different product categories across regions.
- b. Add labels, legends, and titles to enhance the readability of the plot.

## 13. Correlation Heatmap for Numeric Data

- a. Create a heatmap using `sns.heatmap()` to show correlations between numeric variables such as sales, product prices, and customer age.
- b. Customize the color map and annotations.

## Mini Project 2: Analyzing Fitness Tracker Data with Seaborn

In this project, you will analyze fitness tracker data, including information about daily steps, calories burned, sleep duration, and other fitness parameters. You will use Seaborn to create detailed and customized visualizations to uncover patterns and insights from the data.

Task List:

### 1. Import Libraries and Dataset

- a. Import the necessary libraries (Seaborn, Pandas, NumPy, and Matplotlib).
- b. Load the fitness tracker dataset into a Pandas DataFrame.

### 2. Set a Custom Seaborn Theme

- a. Choose an appropriate Seaborn theme using `sns.set_theme()` (e.g., ticks or darkgrid).
- b. Adjust the background style for clarity.

### 3. Examine the Data

- a. Display the first few rows and check for missing values.
- b. Perform any necessary data cleaning steps (drop or fill missing values).

### 4. Step Count Distribution

- a. Create a histogram of daily step counts using `sns.histplot()`.
- b. Adjust the number of bins and add a KDE line to understand the distribution.

### 5. Calories Burned vs. Steps (Scatter Plot)

- a. Create a scatter plot (`sns.scatterplot()`) to show the relationship between daily steps and calories burned.
- b. Use a color palette to differentiate data points by activity level.

### 6. FacetGrid for Activity Type

- a. Use sns.FacetGrid() to create plots showing step count or calories burned by different activity types.
- b. Customize the layout and figure size for easy comparison.

## 7. Create a Line Plot for Sleep Duration Over Time

- a. Use sns.lineplot() to visualize sleep duration trends over time.
- b. Customize the axis labels and line style.

## 8. Box Plot for Daily Steps by Weekday

- a. Create a box plot showing the distribution of daily steps across different weekdays.
- b. Adjust the figure size and grid settings.

## 9. Pair Plot to Visualize Relationships Between Multiple Variables

- a. Use sns.pairplot() to visualize pairwise relationships between variables like steps, calories, and sleep duration.
- b. Apply hue to categorize by activity level or another relevant variable.

## 10. Customizing Figure Size and Grid Styles

- a. Customize the figure size for multiple plots to improve readability.
- b. Set grid styles for a clean look.

## 11. Create Multi-Panel Plots for Sleep and Steps

- a. Use sns.catplot() to create a multi-panel plot that visualizes the relationship between sleep duration and daily steps across different activity levels.
- b. Adjust the layout to create an easy comparison between panels.

## 12. Save and Export the Visualizations

- a. Save the visualizations to .png and .svg formats with a high DPI (e.g., dpi=300).
- b. Ensure the filenames are descriptive for easy identification.

## 13. Heatmap for Correlation Analysis

- a. Use sns.heatmap() to generate a correlation matrix for numeric variables like steps, calories burned, and sleep duration.
- b. Customize the color palette, annotations, and line widths.

# Scikit-Learn

## Day 68

### Introduction to Scikit-Learn & Data Preprocessing

Scikit-Learn is one of the most popular machine learning libraries in Python. It provides simple and efficient tools for data mining, data preprocessing, and building machine learning models.

#### 1. What is Scikit-Learn? Why Use it for Machine Learning?

##### **Definition:**

Scikit-Learn (also written as sklearn) is an open-source machine learning library in Python that provides simple and efficient tools for machine learning and statistical modeling.

##### **Key Features of Scikit-Learn:**

- **Easy to Use:** Provides a simple and consistent API for implementing ML models.
- **Comprehensive:** Supports supervised and unsupervised learning, feature selection, and preprocessing.
- **Integration with Other Libraries:** Works well with Pandas, NumPy, and Matplotlib.
- **Efficient Performance:** Built on top of NumPy, SciPy, and Matplotlib.

## Why Use Scikit-Learn for Machine Learning?

- It provides pre-built machine learning algorithms like linear regression, decision trees, support vector machines, etc.
- It helps in efficient **data preprocessing** using tools for handling missing values, feature scaling, and encoding categorical variables.
- It includes modules for **model selection, cross-validation, and hyperparameter tuning**.

### 2. Installing Scikit-Learn

To install Scikit-Learn, use the following command in your terminal or command prompt:

```
pip install scikit-learn
```

You can verify the installation using:

```
import sklearn  
print(sklearn.__version__) # Check the installed version
```

### 3. Loading Datasets using `sklearn.datasets`

Scikit-Learn provides built-in datasets like **Iris, Boston Housing, Digits, Wine, and Diabetes** datasets.

## Example: Loading the Iris Dataset

```
from sklearn.datasets import load_iris  
import pandas as pd  
  
# Load the dataset  
iris = load_iris()  
  
# Convert to a DataFrame  
df = pd.DataFrame(iris.data, columns=iris.feature_names)  
  
# Display the first 5 rows  
print(df.head())
```

### Output:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

## 4. Splitting Data (train\_test\_split)

Before training a machine learning model, we need to split our dataset into **training** and **testing** sets.

## Example: Splitting the Iris Dataset

```
from sklearn.model_selection import train_test_split

# Load the dataset
X = df # Features
y = iris.target # Target labels

# Split the data into training (80%) and testing (20%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Print dataset sizes
print("Training set size:", X_train.shape)
print("Testing set size:", X_test.shape)
```

### Output:

Training set size: (120, 4)

Testing set size: (30, 4)

## 5. Handling Missing Values using SimpleImputer

Real-world datasets often contain missing values. We can handle missing data by **imputing** (filling) the missing values with techniques like **mean**, **median**, or **mode**.

## Example: Handling Missing Values

```
import numpy as np
from sklearn.impute import SimpleImputer

# Creating a dataset with missing values
data = np.array([[1, 2, np.nan], [4, np.nan, 6], [7, 8, 9]])
df_missing = pd.DataFrame(data, columns=["A", "B", "C"])
print("Dataset with Missing Values:\n", df_missing)

# Create an Imputer (Replace NaN with column mean)
imputer = SimpleImputer(strategy="mean")
df_imputed = pd.DataFrame(imputer.fit_transform(df_missing),
columns=df_missing.columns)

print("\nDataset after Imputation:\n", df_imputed)
```

### Output:

Dataset with Missing Values:

	A	B	C
0	1.0	2.0	NaN
1	4.0	NaN	6.0
2	7.0	8.0	9.0

Dataset after Imputation:

	A	B	C
0	1.0	2.0	7.5
1	4.0	5.0	6.0
2	7.0	8.0	9.0

**Explanation:**

- Column **C** had a missing value in row 0, which was replaced by the mean of column C:  $(6 + 9) / 2 = 7.5$
- Column **B** had a missing value in row 1, which was replaced by the mean of column B:  $(2 + 8) / 2 = 5$

**6. Feature Scaling using StandardScaler and MinMaxScaler**

Machine learning algorithms work better when features are **scaled** to a common range. There are two common scaling techniques:

- **Standardization (StandardScaler)**: Scales the data to have a mean of **0** and a standard deviation of **1**.
- **Normalization (MinMaxScaler)**: Scales the data between **0 and 1**.

**Example: Standardization using StandardScaler**

```
from sklearn.preprocessing import StandardScaler
# Sample data
data = np.array([[50, 2], [20, 8], [30, 4], [40, 6]])
df_scale = pd.DataFrame(data, columns=["Feature1", "Feature2"])

# Apply StandardScaler
scaler = StandardScaler()
df_standardized = pd.DataFrame(scaler.fit_transform(df_scale),
columns=df_scale.columns)

print("Original Data:\n", df_scale)
print("\nStandardized Data:\n", df_standardized)
```

## Example: Normalization using MinMaxScaler

```
from sklearn.preprocessing import MinMaxScaler  
  
# Apply MinMaxScaler  
scaler = MinMaxScaler()  
df_normalized = pd.DataFrame(scaler.fit_transform(df_scale),  
columns=df_scale.columns)  
  
print("\nNormalized Data:\n", df_normalized)
```

## Output Comparison

Original Data:

	Feature1	Feature2
0	50	2
1	20	8
2	30	4
3	40	6

Standardized Data:

	Feature1	Feature2
0	1.183216	-1.183216
1	-1.507557	1.507557
2	-0.507557	-0.507557
3	0.507557	0.507557

Normalized Data:

	Feature1	Feature2
0	1.0	0.0

```
1  0.0  1.0
2  0.333 0.333
3  0.667 0.667
```

### Explanation:

- **Standardization (StandardScaler):** Converts the data into a standard normal distribution.
- **Normalization (MinMaxScaler):** Scales values between 0 and 1, keeping relative proportions.

### Conclusion

Scikit-Learn provides essential tools for **machine learning and data preprocessing**, including:

- ✓ Loading datasets (`sklearn.datasets`)

- ✓ Splitting datasets (`train_test_split`)

- ✓ Handling missing values (`SimpleImputer`)

- ✓ Feature scaling (`StandardScaler`, `MinMaxScaler`)

These preprocessing steps **improve model performance** and **ensure data is in the right format** before training a machine learning model.

Here are **two real-life mini-projects** related to **Scikit-Learn & Data Preprocessing**, with step-by-step implementation and detailed code explanations.

## Mini Project 1: House Price Prediction - Data Preprocessing

### Project Overview:

A real estate company wants to **predict house prices** based on factors like the number of bedrooms, area size, and location. Before building the model, we must **preprocess the data** to handle missing values, scale features, and split the dataset for training and testing.

### Step 1: Install and Import Required Libraries

```
# Install Scikit-Learn if not installed
```

```
!pip install scikit-learn
```

```
# Import necessary libraries
```

```
import numpy as np
```

```
import pandas as pd
```

```
from sklearn.model_selection import train_test_split
```

```
from sklearn.impute import SimpleImputer
```

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler
```

### Step 2: Load the Dataset

```
# Simulated house price dataset
```

```
data = {
```

```
    'Area (sq ft)': [2000, 1500, 1800, 2200, np.nan, 2500, 2700, 1600, 1400, 2100],
```

```
    'Bedrooms': [3, 2, 3, 4, 3, 5, np.nan, 2, 2, 4],
```

```
    'Price ($1000s)': [500, 350, 450, 600, 400, 750, 800, 300, 280, 650]
```

```
}
```

```
df = pd.DataFrame(data)
```

```
print("Original Dataset:\n", df)
```

### Step 3: Handle Missing Values using SimpleImputer

```
# Define imputer (Replace NaN with column mean)
imputer = SimpleImputer(strategy='mean')

# Apply imputer to dataset
df_imputed = pd.DataFrame(imputer.fit_transform(df), columns=df.columns)

print("\nDataset after Handling Missing Values:\n", df_imputed)
```

### Step 4: Splitting Data into Training and Testing Sets

```
# Features and target variable
X = df_imputed[['Area (sq ft)', 'Bedrooms']] # Features
y = df_imputed['Price ($1000s)'] # Target variable

# Split data (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

print("\nTraining Set Size:", X_train.shape)
print("Testing Set Size:", X_test.shape)
```

### Step 5: Feature Scaling using StandardScaler and MinMaxScaler

```
# Standardization (mean = 0, variance = 1)
scaler_standard = StandardScaler()
X_train_standardized = scaler_standard.fit_transform(X_train)
X_test_standardized = scaler_standard.transform(X_test)
```

```
# Normalization (Scaling between 0 and 1)
scaler_minmax = MinMaxScaler()
X_train_normalized = scaler_minmax.fit_transform(X_train)
X_test_normalized = scaler_minmax.transform(X_test)

print("\nStandardized Data (First 3 rows):\n", X_train_standardized[:3])
print("\nNormalized Data (First 3 rows):\n", X_train_normalized[:3])
```

## Summary

- ✓ Handled missing values in house size and bedroom count
- ✓ Split data into training and testing sets
- ✓ Applied feature scaling (Standardization & Normalization)

## Mini Project 2: Customer Churn Prediction - Data Preprocessing

### Project Overview:

A telecom company wants to **predict customer churn** (whether a customer will leave). The dataset contains customer details like **monthly charges, contract type, and total tenure**. Before building the ML model, we must preprocess the data.

### Step 1: Install and Import Required Libraries

```
!pip install scikit-learn
```

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer
```

996

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler
```

## Step 2: Load the Dataset

```
# Simulated customer churn dataset
data = {
    'Tenure (months)': [12, 24, np.nan, 36, 48, 6, 18, 30, np.nan, 40],
    'Monthly Charges ($)': [70, 50, 60, 90, 80, 40, np.nan, 55, 65, 85],
    'Churn (0 = No, 1 = Yes)': [0, 0, 1, 0, 0, 1, 1, 0, 1, 0]
}

df = pd.DataFrame(data)
print("Original Dataset:\n", df)
```

## Step 3: Handling Missing Values using SimpleImputer

```
# Replace missing values with mean
imputer = SimpleImputer(strategy='mean')
df_imputed = pd.DataFrame(imputer.fit_transform(df), columns=df.columns)

print("\nDataset after Handling Missing Values:\n", df_imputed)
```

## Step 4: Splitting Data into Training and Testing Sets

```
# Features and target variable
X = df_imputed[['Tenure (months)', 'Monthly Charges ($)']]
y = df_imputed['Churn']

# Split data (80% training, 20% testing)
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,  
random_state=42)
```

```
print("\nTraining Set Size:", X_train.shape)  
print("Testing Set Size:", X_test.shape)
```

## Step 5: Feature Scaling using StandardScaler and MinMaxScaler

```
# Standardization  
scaler_standard = StandardScaler()  
X_train_standardized = scaler_standard.fit_transform(X_train)  
X_test_standardized = scaler_standard.transform(X_test)
```

```
# Normalization  
scaler_minmax = MinMaxScaler()  
X_train_normalized = scaler_minmax.fit_transform(X_train)  
X_test_normalized = scaler_minmax.transform(X_test)
```

```
print("\nStandardized Data (First 3 rows):\n", X_train_standardized[:3])  
print("\nNormalized Data (First 3 rows):\n", X_train_normalized[:3])
```

## Summary

- ✓ **Replaced missing values** in tenure and monthly charges
- ✓ **Split data** into training and testing sets
- ✓ **Applied feature scaling** (Standardization & Normalization)

## Real-Life Mini Project: Employee Attrition Prediction - Data Preprocessing

**Project Goal:** A company wants to analyze and predict employee attrition (whether an employee will leave or stay). The dataset includes employee **age, salary, experience, work-life balance score, and attrition status**. The data needs preprocessing before training a machine learning model.

### 13 Tasks

#### Step 1: Understanding Scikit-Learn

1. **What is Scikit-Learn?** - Explain why Scikit-Learn is widely used for Machine Learning.
2. **Installing Scikit-Learn** - Install it using pip install scikit-learn.

#### Step 2: Data Handling

3. **Loading the Employee Attrition Dataset** - Load the dataset using Pandas or sklearn.datasets.
4. **Exploring the Data** - Check for missing values, data types, and distributions.

#### Step 3: Data Preprocessing

5. **Handling Missing Values with SimpleImputer** - Fill missing values in age, salary, or experience columns.
6. **Handling Categorical Features (if any)** - Convert categorical data (like department, gender) using encoding techniques.
7. **Removing Outliers** - Identify and remove extreme values in salary and experience.

#### Step 4: Splitting Data

8. **Defining Features (X) and Target (y)** - Separate independent variables (features) from the dependent variable (attrition status).
9. **Splitting Data into Training and Testing Sets** - Use `train_test_split()` to divide data into 80% training and 20% testing.

#### Step 5: Feature Scaling

10. **Using StandardScaler** - Apply standardization to normalize salary and experience.
11. **Using MinMaxScaler** - Scale numeric columns to a 0-1 range.

#### Step 6: Exporting Processed Data

12. **Checking Preprocessed Data** - Print a few rows to verify preprocessing.
13. **Saving the Cleaned Data for Model Training** - Save the processed dataset as a CSV file for further analysis.

### **Mini Project 1: Student Performance Prediction - Data Preprocessing**

**Project Goal:** A university wants to analyze students' academic performance based on factors like **study hours, attendance, previous grades, parental education, and extracurricular activities**. The dataset requires preprocessing, including handling missing values, feature scaling, and splitting data for predictive modeling.

### **Mini Project 2: Loan Default Prediction - Data Preprocessing**

**Project Goal:** A bank wants to predict whether a customer will default on a loan based on features like **income, loan amount, credit score, employment status, and debt-to-income ratio**. The dataset requires cleaning, handling missing values, and normalizing features to prepare it for machine learning models.

# Day 69

Exploratory Data Analysis (EDA) & Feature Engineering

## 1. Understanding Features & Target Variables

### What Are Features & Target Variables?

- **Features:** Independent variables (input data) that influence the outcome.
- **Target Variable:** The dependent variable (output) we want to predict.

### Example: Predicting House Prices

House Size (sqft)	Bedroom s	Location Score	Price (Target)
2000	3	8	\$250,000
1500	2	7	\$200,000
3000	4	9	\$350,000

- **Features:** House Size, Bedrooms, Location Score
- **Target Variable:** Price

## 2. Encoding Categorical Data

Some machine learning models work only with numbers. We must **convert categorical data** (text) into numerical values.

### Methods:

1. **Label Encoding** – Converts categories into numeric labels.
2. **One-Hot Encoding** – Creates separate columns for each category.

## Example: Encoding "City" Data

City	Label Encoding	One-Hot Encoding (Columns)
New York	0	[1, 0, 0]
Los Angeles	1	[0, 1, 0]
Chicago	2	[0, 0, 1]

## Code Example: LabelEncoder

```
from sklearn.preprocessing import LabelEncoder

cities = ['New York', 'Los Angeles', 'Chicago', 'New York', 'Chicago']
label_encoder = LabelEncoder()
encoded_cities = label_encoder.fit_transform(cities)
print(encoded_cities) # Output: [2 1 0 2 0]
```

## Code Example: OneHotEncoder

```
from sklearn.preprocessing import OneHotEncoder
import numpy as np
cities = np.array(['New York', 'Los Angeles', 'Chicago']).reshape(-1, 1)
onehot_encoder = OneHotEncoder(sparse=False)
encoded_cities = onehot_encoder.fit_transform(cities)
print(encoded_cities)
```

## 3. Feature Selection

Not all features are useful. We **select the best features** that have the most impact.

## Methods:

1. **SelectKBest** – Selects top K best features based on statistical tests.
2. **RFE (Recursive Feature Elimination)** – Eliminates least important features step by step.

### Code Example: SelectKBest

```
from sklearn.feature_selection import SelectKBest, f_classif
from sklearn.datasets import load_iris

data = load_iris()
X, y = data.data, data.target

selector = SelectKBest(score_func=f_classif, k=2) # Select top 2 features
X_new = selector.fit_transform(X, y)
print(X_new.shape) # (150, 2)
```

## 4. Feature Extraction (PCA, LDA)

Feature extraction helps **reduce dimensionality** while keeping important information.

### PCA (Principal Component Analysis)

- Reduces many features into a few principal components.
- Used for dimensionality reduction.

### Code Example: PCA

```
from sklearn.decomposition import PCA
import numpy as np
```

```
X = np.random.rand(100, 5) # 5 features
pca = PCA(n_components=2) # Reduce to 2 dimensions
X_pca = pca.fit_transform(X)
print(X_pca.shape) # (100, 2)
```

## 5. Handling Imbalanced Data (SMOTE)

When we have **imbalanced data** (e.g., 90% positive cases, 10% negative), models may become biased.

### SMOTE (Synthetic Minority Over-sampling Technique)

SMOTE **creates synthetic samples** for the minority class to balance data.

#### Code Example: Handling Imbalanced Data with SMOTE

```
from imblearn.over_sampling import SMOTE
from collections import Counter
import numpy as np

# Fake dataset with imbalanced classes
X = np.random.rand(100, 5)
y = np.array([0]*90 + [1]*10) # 90 '0's and 10 '1's

print("Before SMOTE:", Counter(y)) # {0: 90, 1: 10}

smote = SMOTE(sampling_strategy='auto')
X_resampled, y_resampled = smote.fit_resample(X, y)

print("After SMOTE:", Counter(y_resampled)) # {0: 90, 1: 90}
```

## Summary

Concept	Purpose
Understanding Features & Target	Define inputs & outputs
Encoding Categorical Data	Convert text data to numbers
Feature Selection	Pick important features
Feature Extraction	Reduce dimensionality
Handling Imbalanced Data	Balance dataset with SMOTE

## Mini Project 1: Employee Attrition Prediction

### Problem Statement:

A company wants to analyze **why employees leave (attrition)** based on various features like **salary, job role, department, work hours, and satisfaction score**. The goal is to **identify important factors** influencing attrition.

### Step-by-Step Implementation

#### Step 1: Install Required Libraries

```
pip install pandas numpy seaborn scikit-learn imbalanced-learn
```

## Step 2: Import Required Libraries

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, OneHotEncoder, StandardScaler
from sklearn.feature_selection import SelectKBest, f_classif
from sklearn.decomposition import PCA
from imblearn.over_sampling import SMOTE
```

## Step 3: Load Dataset

```
df = pd.read_csv("employee_attrition.csv")
df.head()
```

### Dataset Columns:

Employee ID	Age	Salary	Job Role	Department	Work Hours	Satisfaction Score	Attrition
101	28	50000	Engineer	IT	40	8	Yes
102	35	70000	Manager	HR	45	6	No

- **Target Variable:** Attrition (Yes/No → Needs encoding)
- **Categorical Features:** Job Role, Department

## Step 4: Encoding Categorical Data

```
# Label Encoding for Attrition (Target Variable)
label_encoder = LabelEncoder()
df["Attrition"] = label_encoder.fit_transform(df["Attrition"]) # Yes → 1, No → 0

# One-Hot Encoding for Job Role and Department
df = pd.get_dummies(df, columns=["Job Role", "Department"], drop_first=True)
```

## Step 5: Feature Selection

```
X = df.drop(columns=["Attrition"]) # Features
y = df["Attrition"] # Target

# Select the top 5 best features
selector = SelectKBest(score_func=f_classif, k=5)
X_new = selector.fit_transform(X, y)
print(X_new.shape)
```

## Step 6: Feature Extraction (PCA)

```
pca = PCA(n_components=2) # Reduce to 2 components
X_pca = pca.fit_transform(X_new)
print(X_pca.shape)
```

## Step 7: Handling Imbalanced Data

```
print("Before SMOTE:", y.value_counts())

smote = SMOTE(sampling_strategy='auto')
X_resampled, y_resampled = smote.fit_resample(X_pca, y)

print("After SMOTE:", y_resampled.value_counts())
```

## Mini Project 2: Loan Approval Prediction

### Problem Statement:

A bank wants to predict whether a loan application will be approved or rejected based on income, credit score, employment status, and loan amount.

### Step-by-Step Implementation

#### Step 1: Install Required Libraries

```
pip install pandas numpy seaborn scikit-learn imbalanced-learn
```

#### Step 2: Import Required Libraries

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
```

```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder, StandardScaler
from sklearn.feature_selection import SelectKBest, f_classif
from sklearn.decomposition import PCA
from imblearn.over_sampling import SMOTE
```

### Step 3: Load Dataset

```
df = pd.read_csv("loan_data.csv")
df.head()
```

Applicant ID	Income	Credit Score	Employment Type	Loan Amount	Loan Term	Loan Status
1001	600 00	750	Salaried	250000	15	Approved
1002	400 00	680	Self-Employed	150000	10	Rejected

- **Target Variable:** Loan Status (**Approved/Rejected** → Needs encoding)
- **Categorical Features:** Employment Type

### Step 4: Encoding Categorical Data

```
# Encode Loan Status (Target Variable)
label_encoder = LabelEncoder()
df["Loan Status"] = label_encoder.fit_transform(df["Loan Status"]) # Approved → 1, Rejected → 0

# One-Hot Encode Employment Type
df = pd.get_dummies(df, columns=["Employment Type"], drop_first=True)
```

## Step 5: Feature Selection

```
X = df.drop(columns=["Loan Status"]) # Features  
y = df["Loan Status"] # Target  
  
# Select top 3 features  
selector = SelectKBest(score_func=f_classif, k=3)  
X_new = selector.fit_transform(X, y)  
print(X_new.shape)
```

## Step 6: Feature Extraction (PCA)

```
pca = PCA(n_components=2) # Reduce to 2 components  
X_pca = pca.fit_transform(X_new)  
print(X_pca.shape)
```

## Step 7: Handling Imbalanced Data

```
print("Before SMOTE:", y.value_counts())  
  
smote = SMOTE(sampling_strategy='auto')  
X_resampled, y_resampled = smote.fit_resample(X_pca, y)  
  
print("After SMOTE:", y_resampled.value_counts())
```

## Real-Life Mini Project: Customer Churn Prediction

### Objective:

A telecom company wants to analyze **customer churn** (customers leaving the service) based on features like **monthly charges, contract type, internet service type, and customer tenure**. The goal is to understand key factors that influence churn and prepare the dataset for machine learning.

### 13 Tasks

1. Load the telecom customer dataset into a Pandas DataFrame and inspect the first few rows.
2. Understand the Features & Target Variable: Identify numerical and categorical features and define the target variable (Churn: Yes/No).
3. Check for Missing Values: Find and analyze missing values in the dataset.
4. Handle Missing Data: Fill missing values in numerical columns with the median and categorical columns with the mode.
5. Encode Categorical Data: Use LabelEncoder for binary categorical features and OneHotEncoder for multi-class categorical features like "Internet Service".
6. Visualize Data Distributions: Use Seaborn histograms and count plots to analyze distributions of numerical and categorical variables.
7. Perform Feature Selection: Use SelectKBest to select the top features contributing to churn.
8. Apply Recursive Feature Elimination (RFE): Rank features by importance using RFE with a Decision Tree classifier.
9. Extract Features Using PCA: Reduce dataset dimensions to 2 or 3 principal components using Principal Component Analysis (PCA) and visualize the result using a scatter plot.

10. Apply Linear Discriminant Analysis (LDA): Use LDA to extract new features and analyze if it improves data separability.
11. Detect & Handle Imbalanced Data: Check churn distribution and balance the dataset using SMOTE (Synthetic Minority Oversampling Technique).
12. Scale the Data: Apply StandardScaler or MinMaxScaler to normalize the dataset for machine learning models.
13. Prepare Final Dataset for Machine Learning: Save the processed dataset in a CSV file, ready for training a classification model.

## Real-Life Mini Project Requirements

### Mini Project 1: Predicting Loan Default Risk

#### Objective:

A bank wants to predict which customers are likely to default on their loans. The dataset contains customer details such as **income, employment type, loan amount, credit history, and marital status**. Perform **EDA and Feature Engineering** to prepare the data for machine learning models.

#### Key Tasks:

- Understand features and identify the target variable (loan default: Yes/No).
- Handle missing values in numerical and categorical columns.
- Encode categorical features like employment type and marital status.
- Perform feature selection using **SelectKBest** and **RFE**.
- Extract features using **PCA and LDA** to improve model efficiency.
- Handle class imbalance using **SMOTE** to improve predictions.

## Mini Project 2: EDA & Feature Engineering for an E-Commerce Sales Dataset

### Objective:

An e-commerce company wants to analyze **customer purchase patterns** and improve sales forecasting. The dataset contains details such as **product category, customer location, price, discount applied, purchase frequency, and review ratings**. Perform **EDA and Feature Engineering** to gain insights and prepare the data for machine learning models.

### Key Tasks:

- Define and explore key features affecting product sales.
- Handle missing values in product details and customer ratings.
- Encode categorical variables like product category and customer location using **LabelEncoder**.
- Perform **feature selection** using **SelectKBest** to determine the most important predictors of sales.
- Use **PCA** to extract key patterns in customer behavior.
- Normalize numerical features like product price and discount percentage using **MinMaxScaler** for improved model performance.

## Day 70

### Supervised Learning – Regression Models

Regression models are used in **Supervised Learning** to predict continuous numerical values based on input features. They help in solving real-world problems like **house price prediction, stock price forecasting, sales predictions, etc.**

## 1. Linear Regression (LinearRegression())

### Definition:

Linear Regression is a simple machine learning algorithm that finds a relationship between **independent variables (X)** and a **dependent variable (Y)** using a straight-line equation:

$$Y = mX + b \quad Y = mX + b$$

where:

- **Y** = Predicted value
- **m** = Slope (coefficient)
- **X** = Input feature
- **b** = Intercept

### Syntax:

```
from sklearn.linear_model import LinearRegression

model = LinearRegression()
model.fit(X_train, y_train) # Training the model
y_pred = model.predict(X_test) # Making predictions
```

### Real-Life Example:

#### ◆ Predicting House Prices based on Area

**Step-by-Step Implementation:**

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression

# Sample dataset: House area (sq ft) vs Price
data = {'Area': [500, 800, 1000, 1200, 1500, 1800],
         'Price': [100000, 150000, 200000, 240000, 280000, 350000]}
df = pd.DataFrame(data)

# Splitting data into X (independent) and y (dependent)
X = df[['Area']]
y = df['Price']

# Splitting into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    random_state=42)

# Creating and training Linear Regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Making predictions
y_pred = model.predict(X_test)

# Visualizing results
plt.scatter(X, y, color='blue', label="Actual Data")
plt.plot(X, model.predict(X), color='red', label="Regression Line")
```

```
plt.xlabel("Area (sq ft)")
plt.ylabel("Price ($)")
plt.legend()
plt.show()
```

## 2. Polynomial Regression (PolynomialFeatures())

### Definition:

Polynomial Regression is an extension of Linear Regression that fits a polynomial curve instead of a straight line. It is useful when the data **does not follow a straight-line pattern**.

$$Y = aX^2 + bX + c$$

### Syntax:

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

poly = PolynomialFeatures(degree=2)
X_poly = poly.fit_transform(X)
model = LinearRegression()
model.fit(X_poly, y)
```

### Real-Life Example:

#### ◆ Predicting Salary Growth Over Time

```
from sklearn.preprocessing import PolynomialFeatures
```

```
# Sample dataset: Experience (years) vs Salary ($)
X = np.array([1, 2, 3, 4, 5, 6]).reshape(-1, 1)
y = np.array([30000, 40000, 50000, 65000, 85000, 110000])

# Applying Polynomial Transformation
poly = PolynomialFeatures(degree=2)
X_poly = poly.fit_transform(X)

# Training the model
model = LinearRegression()
model.fit(X_poly, y)

# Predicting and visualizing
plt.scatter(X, y, color='blue')
plt.plot(X, model.predict(X_poly), color='red')
plt.xlabel("Years of Experience")
plt.ylabel("Salary ($)")
plt.show()
```

### 3. Ridge & Lasso Regression (Ridge(), Lasso())

#### Definition:

- **Ridge Regression** (L2 regularization) prevents overfitting by adding a penalty on large coefficients.
- **Lasso Regression** (L1 regularization) helps in feature selection by shrinking some coefficients to zero.

## Syntax:

```
from sklearn.linear_model import Ridge, Lasso  
  
ridge = Ridge(alpha=1.0)  
ridge.fit(X_train, y_train)  
  
lasso = Lasso(alpha=1.0)  
lasso.fit(X_train, y_train)
```

## Real-Life Example:

### ◆ Predicting Car Prices with Regularization

```
from sklearn.linear_model import Ridge, Lasso  
  
ridge = Ridge(alpha=1.0)  
ridge.fit(X_train, y_train)  
  
lasso = Lasso(alpha=1.0)  
lasso.fit(X_train, y_train)  
  
print("Ridge Regression Coefficients:", ridge.coef_)  
print("Lasso Regression Coefficients:", lasso.coef_)
```

## 4. Decision Tree Regression (DecisionTreeRegressor())

### Definition:

A Decision Tree Regression model splits the dataset into branches based on feature conditions, making it useful for **non-linear relationships**.

### Syntax:

```
from sklearn.tree import DecisionTreeRegressor  
  
model = DecisionTreeRegressor()  
model.fit(X_train, y_train)  
y_pred = model.predict(X_test)
```

### Real-Life Example:

#### Predicting Sales Growth

```
from sklearn.tree import DecisionTreeRegressor  
  
model = DecisionTreeRegressor()  
model.fit(X_train, y_train)  
  
y_pred = model.predict(X_test)  
  
plt.scatter(X, y, color='blue')  
plt.plot(X, model.predict(X), color='red')  
plt.show()
```

## 5. Random Forest Regression (`RandomForestRegressor()`)

### Definition:

Random Forest Regression is an **ensemble method** that builds multiple Decision Trees and averages their predictions for better accuracy.

### Syntax:

```
from sklearn.ensemble import RandomForestRegressor  
  
model = RandomForestRegressor(n_estimators=100)  
model.fit(X_train, y_train)  
y_pred = model.predict(X_test)
```

### Real-Life Example:

#### Predicting House Prices with Random Forest

```
from sklearn.ensemble import RandomForestRegressor  
  
model = RandomForestRegressor(n_estimators=100)  
model.fit(X_train, y_train)  
  
y_pred = model.predict(X_test)  
  
print("Predicted House Prices:", y_pred)
```

## 6. Evaluating Regression Models (r2\_score, MSE, MAE)

### Definition:

- **R<sup>2</sup> Score (r2\_score):** Measures how well the model fits the data (1 = perfect fit).
- **Mean Squared Error (MSE):** Penalizes large errors by squaring them.
- **Mean Absolute Error (MAE):** Measures average absolute errors.

### Syntax:

```
from sklearn.metrics import r2_score, mean_squared_error,
mean_absolute_error
```

```
r2 = r2_score(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)

print("R2 Score:", r2)
print("MSE:", mse)
print("MAE:", mae)
```

### Summary of Regression Models:

Model Type	Best Used For	Handles Non-Linearity?
Linear Regression	Simple relationships	✗ No
Polynomial Regression	Curved relationships	✓ Yes
Ridge & Lasso Regression	Regularization, feature selection	✗ No

<b>Decision Tree Regression</b>	Non-linear relationships	✓ Yes
<b>Random Forest Regression</b>	Complex patterns & high accuracy	✓ Yes

## Conclusion

Regression models are crucial for **predicting continuous values** in real-world applications like **house prices, salary prediction, and sales forecasting**.

Let me know if you need **more in-depth explanations or projects!** 🎉😊

## Mini Project 1: House Price Prediction Using Regression Models

### Project Requirement:

Predict house prices based on various features using multiple regression models:

- **Linear Regression**
- **Polynomial Regression**
- **Ridge & Lasso Regression**
- **Decision Tree Regression**
- **Random Forest Regression**
- **Model Evaluation using  $r^2$  Score, MSE, MAE**

## Step-by-Step Implementation

### Step 1: Import Necessary Libraries

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

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.preprocessing import PolynomialFeatures
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import r2_score, mean_squared_error,
mean_absolute_error
```

### Step 2: Load Dataset (House Prices Data)

```
# Creating a sample dataset: Features -> ['Size (sq ft)', 'Bedrooms', 'Age']
data = {
    'Size': [1400, 1600, 1700, 1875, 1100, 1550, 2350, 2450, 1425, 1700],
    'Bedrooms': [3, 3, 3, 4, 2, 3, 4, 4, 3, 3],
    'Age': [20, 15, 18, 10, 25, 12, 8, 5, 20, 18],
    'Price': [245000, 312000, 279000, 308000, 199000, 219000, 405000, 450000,
255000, 280000]
}
df = pd.DataFrame(data)
```

```
# Splitting Features (X) and Target Variable (y)
X = df[['Size', 'Bedrooms', 'Age']]
y = df['Price']

# Splitting into Training and Testing Data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

### Step 3: Apply Multiple Regression Models

#### (A) Linear Regression

```
lr_model = LinearRegression()
lr_model.fit(X_train, y_train)
y_pred_lr = lr_model.predict(X_test)
```

#### (B) Polynomial Regression

```
poly = PolynomialFeatures(degree=2)
X_poly_train = poly.fit_transform(X_train)
X_poly_test = poly.transform(X_test)

poly_model = LinearRegression()
poly_model.fit(X_poly_train, y_train)
y_pred_poly = poly_model.predict(X_poly_test)
```

## (C) Ridge &amp; Lasso Regression

```
ridge_model = Ridge(alpha=1.0)
ridge_model.fit(X_train, y_train)
y_pred_ridge = ridge_model.predict(X_test)
```

```
lasso_model = Lasso(alpha=1.0)
lasso_model.fit(X_train, y_train)
y_pred_lasso = lasso_model.predict(X_test)
```

## (D) Decision Tree Regression

```
dt_model = DecisionTreeRegressor()
dt_model.fit(X_train, y_train)
y_pred_dt = dt_model.predict(X_test)
```

## (E) Random Forest Regression

```
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)
y_pred_rf = rf_model.predict(X_test)
```

**Step 4: Model Evaluation**

```
models = {
    "Linear Regression": y_pred_lr,
    "Polynomial Regression": y_pred_poly,
    "Ridge Regression": y_pred_ridge,
```

```

    "Lasso Regression": y_pred_lasso,
    "Decision Tree Regression": y_pred_dt,
    "Random Forest Regression": y_pred_rf
}

for name, pred in models.items():
    print(f"\n{name}:")
    print("R2 Score:", r2_score(y_test, pred))
    print("MSE:", mean_squared_error(y_test, pred))
    print("MAE:", mean_absolute_error(y_test, pred))

```

## Step 5: Visualizing Model Performance

```

plt.figure(figsize=(10, 5))
plt.scatter(y_test, y_pred_rf, color='blue', label="Random Forest")
plt.scatter(y_test, y_pred_lr, color='red', label="Linear Regression")
plt.scatter(y_test, y_pred_poly, color='green', label="Polynomial Regression")
plt.xlabel("Actual Prices")
plt.ylabel("Predicted Prices")
plt.legend()
plt.show()

```

## Expected Outcome:

- The model with the highest **r<sup>2</sup> score** and lowest **MSE/MAE** is the best predictor of house prices.

- **Random Forest Regression** and **Polynomial Regression** often outperform others in complex datasets.

## Mini Project 2: Predicting Car Prices Based on Features

### Project Requirement:

Build a **car price prediction model** using various regression techniques:

- **Linear Regression**
- **Polynomial Regression**
- **Ridge & Lasso Regression**
- **Decision Tree Regression**
- **Random Forest Regression**
- **Model Evaluation using  $r^2$  Score, MSE, MAE**

### Step-by-Step Implementation

#### Step 1: Import Required Libraries

(Same as Mini Project 1)

#### Step 2: Load & Prepare Dataset

```
# Creating a sample dataset
data = {
    'Horsepower': [130, 250, 190, 300, 210, 150, 170, 190, 200, 220],
    'Mileage': [50000, 20000, 35000, 10000, 25000, 60000, 45000, 32000, 28000,
22000],
    'Age': [5, 2, 3, 1, 3, 6, 5, 4, 2, 3],
    'Price': [15000, 35000, 24000, 45000, 32000, 12000, 18000, 25000, 29000,
```

```
31000]
}
df = pd.DataFrame(data)

# Splitting Features (X) and Target Variable (y)
X = df[['Horsepower', 'Mileage', 'Age']]
y = df['Price']

# Splitting into Training and Testing Data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

### Step 3: Apply Regression Models

(Same as Mini Project 1 but using the df dataset from above.)

### Step 4: Model Evaluation

(Same as Mini Project 1, evaluating r<sup>2</sup> score, MSE, and MAE.)

### Step 5: Visualizing Results

```
plt.figure(figsize=(10, 5))
plt.scatter(y_test, y_pred_rf, color='blue', label="Random Forest")
plt.scatter(y_test, y_pred_lr, color='red', label="Linear Regression")
plt.scatter(y_test, y_pred_poly, color='green', label="Polynomial Regression")
plt.xlabel("Actual Car Prices")
plt.ylabel("Predicted Car Prices")
plt.legend()
plt.show()
```

### Expected Outcome:

- **Random Forest Regression** is likely to provide the most accurate predictions.
- **Polynomial Regression** may also perform well for non-linear trends.

## Real-Life Mini Project: Predicting Salary Based on Experience and Skills

### Project Objective:

Develop a machine learning model to predict **employee salary** based on years of experience, skills, education level, and company size using various **regression models**.

### Project Tasks

#### Task 1: Define the Problem Statement

- Understand the objective of predicting salary using regression models.
- Identify key **independent variables** (features) affecting salary.

#### Task 2: Collect and Explore the Dataset

- Use a sample dataset with **experience, education level, company size, and skills**.
- Perform **EDA (Exploratory Data Analysis)** on salary trends.

#### Task 3: Preprocess the Data

- Handle missing values using **SimpleImputer**.
- Convert categorical features (education level, skills) using **OneHotEncoder** or **LabelEncoder**.

### Task 4: Split Data into Training and Testing Sets

- Use `train_test_split()` to create an **80-20 split** of training and testing data.

### Task 5: Apply Linear Regression

- Train a `LinearRegression()` model.
- Predict salary and evaluate the **r<sup>2</sup> score, MSE, and MAE**.

### Task 6: Implement Polynomial Regression

- Use `PolynomialFeatures()` to transform features into polynomial terms.
- Train the model and compare its performance with linear regression.

### Task 7: Use Ridge and Lasso Regression

- Apply `Ridge()` and `Lasso()` to reduce overfitting.
- Tune the `alpha` parameter and analyze model improvements.

### Task 8: Train a Decision Tree Regression Model

- Train a `DecisionTreeRegressor()` model.
- Compare its results with previous models.

### Task 9: Implement Random Forest Regression

- Train a `RandomForestRegressor()` model with multiple trees.
- Evaluate feature importance.

### Task 10: Compare Model Performance

- Evaluate all models using **r<sup>2</sup> score, MSE, and MAE**.
- Identify which model provides the best salary predictions.

### Task 11: Tune Hyperparameters for the Best Model

- Use **GridSearchCV** to optimize hyperparameters for the best-performing model.
- Retrain and evaluate improvements.

### Task 12: Visualize the Model Predictions

- Plot **actual vs. predicted salaries**.
- Use scatter plots and bar charts to show performance differences.

### Task 13: Deploy the Model (Optional)

- Convert the trained model into a **Flask API or Streamlit app**.
- Allow users to input **experience, skills, and education level** to get salary predictions.

### Expected Outcome:

- A **machine learning model** that accurately predicts **salary** based on various features.
- Comparison of **different regression models** to choose the most effective one.

## Mini Project 1: Predicting Electricity Consumption Using Regression Models

**Objective:** Build a regression model to predict household **electricity consumption** based on factors such as **temperature, number of occupants, appliance usage, and time of day**.

### Key Tasks:

- Collect and preprocess electricity consumption data.
- Perform feature engineering (handling missing values, scaling).
- Train and evaluate **Linear, Polynomial, Ridge, Lasso, Decision Tree, and Random Forest Regression models**.
- Compare model performance using **r<sup>2</sup> score, MSE, and MAE**.
- Optimize the best model and visualize trends.

## Mini Project 2: Forecasting Daily Bike Rentals Using Regression Models

**Objective:** Develop a model to predict **daily bike rental demand** based on historical rental data, **weather conditions, time of year, holidays, and working days**.

### Key Tasks:

- Load and preprocess the dataset.
- Perform feature selection and encoding for categorical data (weather, season).
- Train **multiple regression models (Linear, Polynomial, Ridge, Lasso, Decision Tree, and Random Forest)**.
- Evaluate and compare models using **r<sup>2</sup> score, MSE, and MAE**.
- Optimize the best-performing model and visualize the predictions.

# Day 71

## Supervised Learning – Classification Models

### Introduction to Classification Models

In **Supervised Learning – Classification Models**, the goal is to predict categorical outcomes (labels) based on input features. Unlike regression, which predicts continuous values, **classification** assigns data points to discrete categories (e.g., spam or not spam, loan approved or not approved).

### Key Classification Algorithms

1. **Logistic Regression** – Used for binary classification.
2. **Decision Tree Classifier** – Tree-based model for classification.
3. **Random Forest Classifier** – Ensemble of multiple decision trees.
4. **Support Vector Machine (SVM)** – Separates classes using a hyperplane.
5. **K-Nearest Neighbors (KNN)** – Classifies based on the majority class of nearest data points.
6. **Model Evaluation Metrics** – Accuracy, Precision, Recall, F1-score.

#### 1. Logistic Regression

##### Definition

Logistic Regression is a linear model used for **binary classification**. It estimates probabilities using the **sigmoid function**.

## Syntax

```
from sklearn.linear_model import LogisticRegression  
  
model = LogisticRegression()  
model.fit(X_train, y_train)  
y_pred = model.predict(X_test)
```

## Real-Life Example – Email Spam Classification

**Problem Statement:** Classify emails as **spam** or **not spam** based on their content.

Step-by-Step Solution

### 1. Load Dataset

```
import pandas as pd  
from sklearn.model_selection import train_test_split  
from sklearn.feature_extraction.text import TfidfVectorizer  
  
data = pd.read_csv('emails.csv')  
X = data['text'] # Email content  
y = data['label'] # Spam (1) or Not Spam (0)  
  
vectorizer = TfidfVectorizer()  
X = vectorizer.fit_transform(X)  
  
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,  
random_state=42)
```

## 2. Train Logistic Regression Model

```
from sklearn.linear_model import LogisticRegression  
from sklearn.metrics import accuracy_score  
  
model = LogisticRegression()  
model.fit(X_train, y_train)  
y_pred = model.predict(X_test)  
  
accuracy = accuracy_score(y_test, y_pred)  
print(f'Accuracy: {accuracy:.2f}')
```

## 2. Decision Tree Classifier

### Definition

A **Decision Tree** classifies data by **splitting features into branches** based on conditions.

### Syntax

```
from sklearn.tree import DecisionTreeClassifier  
  
model = DecisionTreeClassifier()  
model.fit(X_train, y_train)  
y_pred = model.predict(X_test)
```

## Real-Life Example – Loan Approval Prediction

**Problem Statement:** Predict if a loan application will be **approved or rejected**.

## Step-by-Step Solution

### 1. Load and Prepare Data

```
data = pd.read_csv('loan_data.csv')
X = data.drop('loan_status', axis=1)
y = data['loan_status']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

### 2. Train Decision Tree Model

```
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier()
model.fit(X_train, y_train)
```

### 3. Make Predictions & Evaluate

```
y_pred = model.predict(X_test)
from sklearn.metrics import accuracy_score
print(f'Accuracy: {accuracy_score(y_test, y_pred):.2f}')
```

## 3. Random Forest Classifier

### Definition

Random Forest is an **ensemble learning method** that combines multiple decision trees.

## Syntax

```
from sklearn.ensemble import RandomForestClassifier  
  
model = RandomForestClassifier()  
model.fit(X_train, y_train)  
y_pred = model.predict(X_test)
```

## Real-Life Example – Disease Diagnosis

**Problem Statement:** Predict whether a **patient has diabetes** based on health parameters.

Step-by-Step Solution

### 1. Load and Prepare Data

```
data = pd.read_csv('diabetes.csv')  
X = data.drop('diabetes', axis=1)  
y = data['diabetes']  
  
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,  
random_state=42)
```

### 2. Train Random Forest Model

```
model = RandomForestClassifier(n_estimators=100, random_state=42)  
model.fit(X_train, y_train)
```

### 3. Evaluate Performance

```
y_pred = model.predict(X_test)
print(f'Accuracy: {accuracy_score(y_test, y_pred):.2f}')
```

## 4. Support Vector Machine (SVM)

### Definition

SVM finds an **optimal hyperplane** to separate different classes.

### Syntax

```
from sklearn.svm import SVC
```

```
model = SVC()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

## Real-Life Example – Sentiment Analysis

**Problem Statement:** Classify tweets as **positive or negative**.

### Step-by-Step Solution

#### 1. Load and Prepare Data

```
data = pd.read_csv('tweets.csv')
X = data['text']
y = data['sentiment']

vectorizer = TfidfVectorizer()
```

```
X = vectorizer.fit_transform(X)
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,  
random_state=42)
```

## 2. Train SVM Model

```
model = SVC()  
model.fit(X_train, y_train)
```

## 3. Evaluate Performance

```
y_pred = model.predict(X_test)  
print(f'Accuracy: {accuracy_score(y_test, y_pred):.2f}')
```

## 5. K-Nearest Neighbors (KNN)

### Definition

KNN classifies data based on the **majority class** of its nearest neighbors.

### Syntax

```
from sklearn.neighbors import KNeighborsClassifier
```

```
model = KNeighborsClassifier(n_neighbors=5)  
model.fit(X_train, y_train)  
y_pred = model.predict(X_test)
```

## Real-Life Example – Customer Churn Prediction

**Problem Statement:** Predict if a customer will **churn (leave the company) or stay.**

Step-by-Step Solution

### 1. Load and Prepare Data

```
data = pd.read_csv('customer_churn.csv')
X = data.drop('churn', axis=1)
y = data['churn']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

### 2. Train KNN Model

```
model = KNeighborsClassifier(n_neighbors=5)
model.fit(X_train, y_train)
```

### 3. Evaluate Performance

```
y_pred = model.predict(X_test)
print(f'Accuracy: {accuracy_score(y_test, y_pred):.2f}')
```

## 6 Evaluating Classification Models

We use **accuracy, precision, recall, and F1-score** to evaluate models.

## Syntax

```
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score

print(f'Accuracy: {accuracy_score(y_test, y_pred):.2f}')
print(f'Precision: {precision_score(y_test, y_pred, average="binary"):.2f}')
print(f'Recall: {recall_score(y_test, y_pred, average="binary"):.2f}')
print(f'F1 Score: {f1_score(y_test, y_pred, average="binary"):.2f}')
```

## Conclusion

These models are widely used in **finance, healthcare, marketing, and fraud detection**. Would you like a detailed **comparison of these models with real-world datasets?** 

## Mini Project 1: Customer Churn Prediction

### Problem Statement

A telecommunications company wants to predict whether a customer will **churn (leave the company) or stay** based on various customer features.

### Step-by-Step Implementation

#### Step 1: Install Required Libraries

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

```
import seaborn as sns

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, classification_report
```

## Step 2: Load the Dataset

```
data = pd.read_csv("customer_churn.csv") # Load dataset

# Display first five rows
print(data.head())
```

### Dataset Columns:

- customer\_id – Unique ID for customers.
- tenure – Number of months with the company.
- monthly\_charges – Amount paid per month.
- total\_charges – Total amount paid.
- contract\_type – Type of contract (Monthly, Yearly, etc.).
- internet\_service – Type of internet (DSL, Fiber, No).
- payment\_method – Mode of payment.
- churn – Target variable (Yes = 1, No = 0).

### Step 3: Data Preprocessing

```
# Drop customer_id column
data.drop(columns=['customer_id'], inplace=True)

# Convert categorical variables to numerical
label_enc = LabelEncoder()
data['contract_type'] = label_enc.fit_transform(data['contract_type'])
data['internet_service'] = label_enc.fit_transform(data['internet_service'])
data['payment_method'] = label_enc.fit_transform(data['payment_method'])
data['churn'] = label_enc.fit_transform(data['churn']) # Target variable
```

### Step 4: Split Data

```
X = data.drop(columns=['churn']) # Features
y = data['churn'] # Target variable

# Split into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

### Step 5: Train Classification Models

#### 1. Logistic Regression

```
log_reg = LogisticRegression()
log_reg.fit(X_train, y_train)
y_pred_log = log_reg.predict(X_test)
```

## 2. Decision Tree Classifier

```
dt_clf = DecisionTreeClassifier()  
dt_clf.fit(X_train, y_train)  
y_pred_dt = dt_clf.predict(X_test)
```

## 3. Random Forest Classifier

```
rf_clf = RandomForestClassifier(n_estimators=100)  
rf_clf.fit(X_train, y_train)  
y_pred_rf = rf_clf.predict(X_test)
```

## 4. Support Vector Machine (SVM)

```
svm_clf = SVC()  
svm_clf.fit(X_train, y_train)  
y_pred_svm = svm_clf.predict(X_test)
```

## 5. K-Nearest Neighbors (KNN)

```
knn_clf = KNeighborsClassifier(n_neighbors=5)  
knn_clf.fit(X_train, y_train)  
y_pred_knn = knn_clf.predict(X_test)
```

## Step 6: Model Evaluation

```
models = {
    "Logistic Regression": y_pred_log,
    "Decision Tree": y_pred_dt,
    "Random Forest": y_pred_rf,
    "SVM": y_pred_svm,
    "KNN": y_pred_knn
}

for model_name, y_pred in models.items():
    print(f"\n{model_name} Performance:")
    print(f"Accuracy: {accuracy_score(y_test, y_pred):.2f}")
    print(f"Precision: {precision_score(y_test, y_pred):.2f}")
    print(f"Recall: {recall_score(y_test, y_pred):.2f}")
    print(f"F1 Score: {f1_score(y_test, y_pred):.2f}")
    print("-" * 50)
```

## Mini Project 2: Loan Approval Prediction

### Problem Statement

A bank wants to automate its loan approval process by predicting whether a customer will be **granted a loan (1) or denied (0)** based on past records.

### Step-by-Step Implementation

#### Step 1: Install Required Libraries

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

```
import seaborn as sns

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, classification_report
```

## Step 2: Load the Dataset

```
data = pd.read_csv("loan_data.csv") # Load dataset

# Display first five rows
print(data.head())
```

### Dataset Columns:

- applicant\_income – Monthly income of applicant.
- loan\_amount – Loan amount requested.
- credit\_history – Credit score history (1 = Good, 0 = Bad).
- loan\_term – Duration of loan (months).
- married – Marital status (Yes/No).
- self\_employed – Self-employment status (Yes/No).
- loan\_approved – Target variable (1 = Approved, 0 = Denied).

### Step 3: Data Preprocessing

```
# Convert categorical variables to numerical
label_enc = LabelEncoder()
data['married'] = label_enc.fit_transform(data['married'])
data['self_employed'] = label_enc.fit_transform(data['self_employed'])
data['loan_approved'] = label_enc.fit_transform(data['loan_approved']) # Target
variable
```

### Step 4: Split Data

```
X = data.drop(columns=['loan_approved']) # Features
y = data['loan_approved'] # Target variable
```

```
# Split into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

### Step 5: Train Classification Models

#### 1. Logistic Regression

```
log_reg = LogisticRegression()
log_reg.fit(X_train, y_train)
y_pred_log = log_reg.predict(X_test)
```

## 2. Decision Tree Classifier

```
dt_clf = DecisionTreeClassifier()  
dt_clf.fit(X_train, y_train)  
y_pred_dt = dt_clf.predict(X_test)
```

## 3. Random Forest Classifier

```
rf_clf = RandomForestClassifier(n_estimators=100)  
rf_clf.fit(X_train, y_train)  
y_pred_rf = rf_clf.predict(X_test)
```

## 4. Support Vector Machine (SVM)

```
svm_clf = SVC()  
svm_clf.fit(X_train, y_train)  
y_pred_svm = svm_clf.predict(X_test)
```

## 5. K-Nearest Neighbors (KNN)

```
knn_clf = KNeighborsClassifier(n_neighbors=5)  
knn_clf.fit(X_train, y_train)  
y_pred_knn = knn_clf.predict(X_test)
```

## Step 6: Model Evaluation

```

models = {
    "Logistic Regression": y_pred_log,
    "Decision Tree": y_pred_dt,
    "Random Forest": y_pred_rf,
    "SVM": y_pred_svm,
    "KNN": y_pred_knn
}

for model_name, y_pred in models.items():
    print(f"\n{model_name} Performance:")
    print(f"Accuracy: {accuracy_score(y_test, y_pred):.2f}")
    print(f"Precision: {precision_score(y_test, y_pred):.2f}")
    print(f"Recall: {recall_score(y_test, y_pred):.2f}")
    print(f"F1 Score: {f1_score(y_test, y_pred):.2f}")
    print("-" * 50)

```

## Mini Project: Employee Promotion Prediction

### Problem Statement:

A company wants to automate the **employee promotion process** by predicting whether an employee should be promoted or not based on past records. The goal is to build a classification model using **Supervised Learning – Classification Models.**

## Tasks

1. **Load the dataset:** Import employee data containing details like experience, performance rating, training scores, department, etc.
2. **Understand features & target variable:** Identify which columns are relevant for prediction and check data types.
3. **Handle missing values:** Check for missing values and decide whether to fill them (imputation) or drop the rows.
4. **Encoding categorical variables:** Convert categorical features (e.g., department, education level) into numerical values using techniques like **One-Hot Encoding or Label Encoding**.
5. **Feature scaling:** Apply **StandardScaler** or **MinMaxScaler** to normalize numerical features like salary, training scores, and experience.
6. **Split data into training and testing sets:** Use **train\_test\_split()** to divide the dataset (e.g., 80% training, 20% testing).
7. **Train a Logistic Regression model:** Fit the model using **LogisticRegression()** and make predictions.
8. **Train Decision Tree and Random Forest models:** Compare the performance of **DecisionTreeClassifier()** and **RandomForestClassifier()** models.
9. **Train Support Vector Machine (SVM) and KNN models:** Implement **SVC()** and **KNeighborsClassifier()** to see how they perform.
10. **Evaluate models using accuracy, precision, recall, and F1-score:** Use **accuracy\_score**, **precision\_score**, **recall\_score**, and **f1\_score** to compare model performance.
11. **Hyperparameter tuning:** Use **GridSearchCV** or **RandomizedSearchCV** to optimize hyperparameters for better performance.
12. **Feature selection:** Apply **SelectKBest** or **Recursive Feature Elimination (RFE)** to choose the most important features.
13. **Final model deployment:** Save the best-performing model using **joblib** or **pickle** for real-world use in an HR system.

## Mini Project 1: Disease Prediction Using Patient Data

Problem Statement:

A healthcare company wants to develop a system to predict whether a patient is likely to have a certain disease (e.g., diabetes, heart disease) based on medical data such as **age, BMI, blood pressure, glucose level, and lifestyle habits**. Using **Supervised Learning – Classification Models**, the goal is to classify patients as **diseased or healthy**.

## Mini Project 2: Email Spam Detection

Problem Statement:

An email service provider wants to build a **spam detection system** that can classify incoming emails as **spam or not spam** based on features like **word frequency, sender reputation, and email structure**. Using **Supervised Learning – Classification Models**, the goal is to **train a model that automatically filters spam emails**.

# Day 72

## Model Optimization & Hyperparameter Tuning

In machine learning, **model optimization and hyperparameter tuning** are crucial steps to improve model accuracy and performance. These techniques help in finding the best parameters, reducing overfitting, and ensuring the model generalizes well on unseen data.

## 1. Cross-Validation (`cross_val_score()`)

### Definition:

Cross-validation is a resampling technique used to evaluate the performance of a model by splitting the dataset into multiple subsets for training and testing. It helps reduce overfitting and ensures that the model performs well on unseen data.

### Syntax:

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(model, X, y, cv=5) # 5-fold cross-validation
print("Cross-validation scores:", scores)
print("Mean score:", scores.mean())
```

### Real-Life Example: Credit Score Classification

Imagine a bank wants to predict whether a customer will repay a loan. Using cross-validation ensures the model performs well across different subsets of customer data.

### Step-by-Step Implementation:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier

# Load dataset
iris = load_iris()
X, y = iris.data, iris.target
```

```
# Initialize model
model = RandomForestClassifier()

# Perform cross-validation
scores = cross_val_score(model, X, y, cv=5)

# Print results
print("Cross-validation scores:", scores)
print("Mean score:", scores.mean())
```

## 2. Grid Search (GridSearchCV)

### Definition:

Grid Search is an exhaustive hyperparameter tuning technique that tests all possible combinations of hyperparameters to find the best model configuration.

### Syntax:

```
from sklearn.model_selection import GridSearchCV

param_grid = {'n_estimators': [50, 100, 200], 'max_depth': [3, 5, 10]}
grid_search = GridSearchCV(RandomForestClassifier(), param_grid, cv=5)
grid_search.fit(X, y)

print("Best parameters:", grid_search.best_params_)
print("Best score:", grid_search.best_score_)
```

## Real-Life Example: Predicting Customer Churn

A telecom company wants to predict customer churn. Using Grid Search, we find the best hyperparameters for maximum accuracy.

### Step-by-Step Implementation:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import GridSearchCV

# Load dataset
iris = load_iris()
X, y = iris.data, iris.target

# Define hyperparameter grid
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [3, 5, 10]
}

# Perform Grid Search
grid_search = GridSearchCV(RandomForestClassifier(), param_grid, cv=5)
grid_search.fit(X, y)

# Print best parameters
print("Best parameters:", grid_search.best_params_)
print("Best score:", grid_search.best_score_)
```

### 3. Randomized Search (RandomizedSearchCV)

#### Definition:

Randomized Search is an optimization technique similar to Grid Search, but instead of checking all combinations, it randomly selects parameter values from a predefined range, making it more efficient.

#### Syntax:

```
from sklearn.model_selection import RandomizedSearchCV
```

```
param_dist = {'n_estimators': [50, 100, 200], 'max_depth': [3, 5, 10]}
random_search = RandomizedSearchCV(RandomForestClassifier(), param_dist,
n_iter=5, cv=5)
random_search.fit(X, y)
```

```
print("Best parameters:", random_search.best_params_)
print("Best score:", random_search.best_score_)
```

#### Real-Life Example: Fraud Detection System

A financial institution wants to detect fraudulent transactions. Randomized Search helps speed up hyperparameter tuning for better fraud detection.

#### Step-by-Step Implementation:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import RandomizedSearchCV

# Load dataset
```

```
iris = load_iris()
X, y = iris.data, iris.target

# Define hyperparameter distribution
param_dist = {
    'n_estimators': [50, 100, 200],
    'max_depth': [3, 5, 10]
}

# Perform Randomized Search
random_search = RandomizedSearchCV(RandomForestClassifier(), param_dist,
n_iter=5, cv=5)
random_search.fit(X, y)

# Print best parameters
print("Best parameters:", random_search.best_params_)
print("Best score:", random_search.best_score_)
```

#### 4. Feature Importance Analysis

##### **Definition:**

Feature importance analysis helps identify which features contribute the most to a model's predictions. It helps in reducing dimensionality and improving model efficiency.

##### **Syntax:**

```
from sklearn.ensemble import RandomForestClassifier
```

```
model = RandomForestClassifier()  
model.fit(X, y)  
  
importances = model.feature_importances_  
print("Feature importances:", importances)
```

### Real-Life Example: Real Estate Price Prediction

A real estate company wants to predict house prices. Feature importance helps determine which factors (location, size, number of bedrooms) have the biggest impact.

#### Step-by-Step Implementation:

```
from sklearn.ensemble import RandomForestClassifier  
from sklearn.datasets import load_iris  
  
# Load dataset  
iris = load_iris()  
X, y = iris.data, iris.target  
  
# Train model  
model = RandomForestClassifier()  
model.fit(X, y)  
  
# Get feature importances  
importances = model.feature_importances_  
  
# Print feature importances  
for feature, importance in zip(iris.feature_names, importances):
```

```
print(f"feature}: {importance:.4f}")
```

## 5. Avoiding Overfitting (Regularization, Dropout)

### Definition:

Overfitting occurs when a model learns too much from the training data and performs poorly on unseen data. Techniques like **regularization (L1/L2 penalty)** and **dropout (in neural networks)** help prevent this.

### Regularization Syntax:

```
from sklearn.linear_model import Ridge, Lasso
```

```
ridge = Ridge(alpha=1.0)  
lasso = Lasso(alpha=0.1)
```

```
ridge.fit(X, y)  
lasso.fit(X, y)
```

### Real-Life Example: Predicting Sales Revenue

A retail company wants to predict monthly sales revenue. Regularization helps in **preventing overfitting** and improving generalization.

### Step-by-Step Implementation:

```
from sklearn.linear_model import Ridge, Lasso  
from sklearn.datasets import load_diabetes  
from sklearn.model_selection import train_test_split
```

```
# Load dataset
diabetes = load_diabetes()
X_train, X_test, y_train, y_test = train_test_split(diabetes.data, diabetes.target,
test_size=0.2, random_state=42)

# Train Ridge and Lasso models
ridge = Ridge(alpha=1.0)
lasso = Lasso(alpha=0.1)

ridge.fit(X_train, y_train)
lasso.fit(X_train, y_train)

# Print coefficients
print("Ridge coefficients:", ridge.coef_)
print("Lasso coefficients:", lasso.coef_)
```

## Conclusion

Model optimization and hyperparameter tuning are crucial for improving model performance. Techniques like **cross-validation**, **GridSearchCV**, and **RandomizedSearchCV** help in finding the best model configurations, while **feature importance analysis** and **regularization** prevent overfitting.

Would you like any modifications or explanations on specific parts? 

## Mini Project 1: Credit Card Fraud Detection (Classification Optimization & Tuning)

### Problem Statement

A bank wants to detect fraudulent credit card transactions. Given transaction details (amount, location, transaction type, etc.), we need to build a classification model and optimize it using **cross-validation, hyperparameter tuning, and feature selection techniques.**

### Step 1: Install and Import Required Libraries

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

from sklearn.model_selection import train_test_split, cross_val_score,
GridSearchCV, RandomizedSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report
```

### Step 2: Load and Explore the Dataset

```
df = pd.read_csv('credit_card_fraud.csv')

# Display first 5 rows
print(df.head())
```

1060

Private & Confidential : Vetri Technology Solutions

```
# Check missing values
print(df.isnull().sum())

# Check class distribution
print(df['fraudulent'].value_counts()) # Target column: 1 = Fraud, 0 = Not Fraud
```

### Step 3: Preprocessing & Feature Engineering

```
# Convert categorical variables to numerical
df = pd.get_dummies(df, drop_first=True)

# Define Features (X) and Target (y)
X = df.drop('fraudulent', axis=1)
y = df['fraudulent']

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42, stratify=y)
```

### Step 4: Apply Cross-Validation

```
model = RandomForestClassifier()
scores = cross_val_score(model, X_train, y_train, cv=5, scoring='accuracy')

print("Cross-validation scores:", scores)
```

```
print("Mean Accuracy:", scores.mean())
```

### Step 5: Perform Grid Search for Hyperparameter Tuning

```
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [5, 10, 15],
    'criterion': ['gini', 'entropy']
}

grid_search = GridSearchCV(RandomForestClassifier(), param_grid, cv=5,
                           scoring='accuracy')
grid_search.fit(X_train, y_train)

print("Best Parameters:", grid_search.best_params_)
print("Best Score:", grid_search.best_score_)
```

### Step 6: Perform Randomized Search for Hyperparameter Tuning

```
param_dist = {
    'n_estimators': [50, 100, 200],
    'max_depth': [5, 10, 15],
    'criterion': ['gini', 'entropy']
}

random_search = RandomizedSearchCV(RandomForestClassifier(), param_dist,
                                    n_iter=5, cv=5, scoring='accuracy', random_state=42)
```

```
random_search.fit(X_train, y_train)

print("Best Parameters (Randomized Search):", random_search.best_params_)
print("Best Score:", random_search.best_score_)
```

## Step 7: Feature Importance Analysis

```
model = RandomForestClassifier(n_estimators=100)
model.fit(X_train, y_train)

# Get feature importances
feature_importance = pd.Series(model.feature_importances_,
index=X.columns).sort_values(ascending=False)
print("Feature Importance:\n", feature_importance)

plt.figure(figsize=(10, 5))
sns.barplot(x=feature_importance, y=feature_importance.index)
plt.xlabel('Importance Score')
plt.ylabel('Features')
plt.title('Feature Importance Analysis')
plt.show()
```

## Step 8: Evaluate the Final Model

```
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)
```

```
print("Final Model Accuracy:", accuracy_score(y_test, y_pred))
print("Classification Report:\n", classification_report(y_test, y_pred))
```

## Mini Project 2: Energy Consumption Prediction (Regression Optimization & Tuning)

### Problem Statement

An energy company wants to predict electricity consumption based on weather conditions, time of day, and other factors. We will build a regression model and optimize it using **cross-validation, hyperparameter tuning, feature selection, and regularization**.

### Step 1: Install and Import Required Libraries

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

from sklearn.model_selection import train_test_split, cross_val_score,
GridSearchCV, RandomizedSearchCV
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import Ridge, Lasso
from sklearn.metrics import mean_absolute_error, mean_squared_error,
r2_score
```

**Step 2: Load and Explore the Dataset**

```
df = pd.read_csv('energy_consumption.csv')

# Display first 5 rows
print(df.head())

# Check missing values
print(df.isnull().sum())

# Basic statistics
print(df.describe())
```

**Step 3: Preprocessing & Feature Engineering**

```
# Convert categorical variables to numerical
df = pd.get_dummies(df, drop_first=True)

# Define Features (X) and Target (y)
X = df.drop('energy_consumed', axis=1) # 'energy_consumed' is the target
variable
y = df['energy_consumed']

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

**Step 4: Apply Cross-Validation**

```
model = RandomForestRegressor()  
scores = cross_val_score(model, X_train, y_train, cv=5, scoring='r2')  
  
print("Cross-validation scores:", scores)  
print("Mean R2 Score:", scores.mean())
```

**Step 5: Hyperparameter Tuning using Grid Search**

```
param_grid = {  
    'n_estimators': [50, 100, 200],  
    'max_depth': [5, 10, 15]  
}  
  
grid_search = GridSearchCV(RandomForestRegressor(), param_grid, cv=5,  
                           scoring='r2')  
grid_search.fit(X_train, y_train)  
  
print("Best Parameters:", grid_search.best_params_)  
print("Best Score:", grid_search.best_score_)
```

**Step 6: Feature Importance Analysis**

```
model = RandomForestRegressor(n_estimators=100)  
model.fit(X_train, y_train)
```

```
feature_importance = pd.Series(model.feature_importances_,  
index=X.columns).sort_values(ascending=False)  
print("Feature Importance:\n", feature_importance)  
  
plt.figure(figsize=(10, 5))  
sns.barplot(x=feature_importance, y=feature_importance.index)  
plt.xlabel('Importance Score')  
plt.ylabel('Features')  
plt.title('Feature Importance Analysis')  
plt.show()
```

### Step 7: Avoid Overfitting using Regularization (Ridge/Lasso)

```
ridge = Ridge(alpha=1.0)  
ridge.fit(X_train, y_train)  
print("Ridge R2 Score:", ridge.score(X_test, y_test))
```

```
lasso = Lasso(alpha=0.1)  
lasso.fit(X_train, y_train)  
print("Lasso Coefficients:", lasso.coef_)
```

### Step 8: Final Model Evaluation

```
best_model = grid_search.best_estimator_  
y_pred = best_model.predict(X_test)  
  
print("R2 Score:", r2_score(y_test, y_pred))
```

```
print("MSE:", mean_squared_error(y_test, y_pred))
print("MAE:", mean_absolute_error(y_test, y_pred))
```

## Mini Project: Customer Churn Prediction (Model Optimization & Hyperparameter Tuning)

### Problem Statement:

A telecom company wants to predict customer churn (whether a customer will leave the company). The dataset contains customer details like tenure, monthly charges, contract type, and whether they have left the company. The goal is to build a classification model and optimize it using various **hyperparameter tuning techniques**.

### Tasks List

#### 1. Load and Explore the Dataset

- Import the necessary libraries and load the dataset.
- Perform initial exploratory data analysis (EDA).

#### 2. Handle Missing Values

- Identify and handle any missing values in the dataset.
- Use appropriate imputation techniques.

#### 3. Convert Categorical Variables

- Encode categorical variables using LabelEncoder or OneHotEncoder.

4. Define Features and Target Variable

- Separate independent features (X) and the target variable (y).

5. Split the Dataset into Training and Testing Sets

- Use `train_test_split()` to divide the data into training and testing sets.

6. Train a Base Classification Model

- Train an initial classification model (e.g., `RandomForestClassifier`).

7. Evaluate the Model using Cross-Validation

- Use `cross_val_score()` to measure model performance with cross-validation.

8. Perform Hyperparameter Tuning using Grid Search

- Define a hyperparameter grid and apply `GridSearchCV` to find the best parameters.

9. Perform Hyperparameter Tuning using Randomized Search

- Use `RandomizedSearchCV` to compare with Grid Search results.

10. Analyze Feature Importance

- Identify important features using feature importance scores from a tree-based model.

11. Reduce Overfitting using Regularization

- Apply Ridge (L2) or Lasso (L1) regularization to reduce overfitting.

## 12. Implement Dropout for Neural Networks (Optional)

- If using deep learning models, apply dropout to prevent overfitting.

## 13. Evaluate the Final Optimized Model

- Measure performance using accuracy, precision, recall, and F1-score.
- Compare the performance before and after tuning.

## **Mini Project 1: Sales Forecasting for an E-commerce Platform**

### **Problem Statement:**

An e-commerce company wants to improve its sales predictions for different product categories. The dataset includes historical sales data, seasonal trends, product prices, and customer purchase behavior. The goal is to build a **regression model** and optimize it using **Grid Search and Randomized Search**. **Cross-validation** will be used to ensure the model's performance, and **feature importance analysis** will help identify the most influential factors in sales.

## **Mini Project 2: Fraud Detection in Online Transactions**

### **Problem Statement:**

A financial company wants to detect fraudulent transactions based on various customer activities. The dataset includes transaction amount, location, time, user behavior, and device details. The objective is to develop a **classification model** and optimize it with **hyperparameter tuning techniques** like **Grid Search and Randomized Search**. **Feature importance analysis** will help identify the key factors contributing to fraudulent transactions, while **regularization techniques** will be used to prevent overfitting.

# Day 73

## Unsupervised Learning – Clustering & Dimensionality Reduction

Unsupervised learning is a type of machine learning where the model is trained on **unlabeled data** and discovers patterns or structures **without explicit supervision**.

### Why Use Unsupervised Learning?

- To **group similar data points** (Clustering)
- To **reduce dimensionality** for better visualization and efficiency
- To **find hidden structures** in data
- To **improve the performance** of supervised learning models by preprocessing data

### 1. K-Means Clustering (KMeans)

K-Means is a **centroid-based clustering algorithm** that divides data into k clusters. It minimizes the distance between data points and cluster centroids.

#### Syntax:

```
from sklearn.cluster import KMeans
```

```
kmeans = KMeans(n_clusters=3, random_state=42)
```

```
kmeans.fit(X)
```

```
labels = kmeans.labels_
```

## Real-Life Example: Customer Segmentation in a Shopping Mall

**Problem:** A mall wants to segment its customers based on annual income and spending score.

### Step-by-Step Implementation

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

# Sample dataset (Annual Income vs Spending Score)
data = np.array([[15, 39], [16, 81], [17, 6], [18, 77], [19, 40],
                 [20, 76], [21, 6], [22, 94], [23, 3], [24, 72]])

# Apply K-Means
kmeans = KMeans(n_clusters=3, random_state=42)
kmeans.fit(data)

# Plot results
plt.scatter(data[:, 0], data[:, 1], c=kmeans.labels_, cmap='viridis')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
            s=300, c='red', marker='X', label='Centroids')
plt.xlabel('Annual Income (k$)')
plt.ylabel('Spending Score (1-100)')
plt.legend()
plt.show()
```

**Outcome:** Customers are divided into 3 groups based on their spending behavior.

## 2. Hierarchical Clustering (AgglomerativeClustering)

Hierarchical clustering builds a tree-like **dendrogram** to cluster data.

### Syntax:

```
from sklearn.cluster import AgglomerativeClustering  
  
hc = AgglomerativeClustering(n_clusters=3)  
hc.fit(X)  
labels = hc.labels_
```

## Real-Life Example: Grouping Students Based on Study Hours

**Problem:** A teacher wants to group students based on the number of hours they study.

### Step-by-Step Implementation

```
import scipy.cluster.hierarchy as sch  
from sklearn.cluster import AgglomerativeClustering  
  
# Sample dataset (Hours Studied vs Exam Score)  
data = np.array([[2, 50], [3, 60], [5, 80], [7, 90], [9, 95]])  
  
# Create Dendrogram  
plt.figure(figsize=(6, 4))  
sch.dendrogram(sch.linkage(data, method='ward'))  
plt.title('Dendrogram')  
plt.xlabel('Students')
```

```
plt.ylabel('Distance')
plt.show()

# Apply Agglomerative Clustering
hc = AgglomerativeClustering(n_clusters=2)
labels = hc.fit_predict(data)

# Plot clusters
plt.scatter(data[:, 0], data[:, 1], c=labels, cmap='rainbow')
plt.xlabel('Hours Studied')
plt.ylabel('Exam Score')
plt.show()
```

**Outcome:** Students are divided into high and low study groups.

### 3. DBSCAN Clustering (DBSCAN)

DBSCAN is a **density-based clustering algorithm** that finds clusters of different shapes and sizes.

#### Syntax:

```
from sklearn.cluster import DBSCAN

dbscan = DBSCAN(eps=0.5, min_samples=5)
dbscan.fit(X)
labels = dbscan.labels_
```

## Real-Life Example: Detecting Outliers in Credit Card Transactions

**Problem:** A bank wants to detect unusual spending patterns.

### Step-by-Step Implementation

```
from sklearn.datasets import make_moons
from sklearn.preprocessing import StandardScaler
import seaborn as sns

# Generate dataset (Moons Shape for better visualization)
X, _ = make_moons(n_samples=300, noise=0.05)

# Scale data
X = StandardScaler().fit_transform(X)

# Apply DBSCAN
dbscan = DBSCAN(eps=0.3, min_samples=5)
labels = dbscan.fit_predict(X)

# Plot clusters
sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=labels, palette="deep")
plt.title("DBSCAN Clustering")
plt.show()
```

**Outcome:** DBSCAN groups dense clusters and identifies outliers.

#### 4. Principal Component Analysis (PCA)

PCA **reduces dimensions** while preserving the most important variance in data.

##### Syntax:

```
from sklearn.decomposition import PCA
```

```
pca = PCA(n_components=2)  
X_pca = pca.fit_transform(X)
```

#### Real-Life Example: Reducing Features in a Stock Market Dataset

**Problem:** A stock trader wants to reduce the number of technical indicators used for analysis.

##### Step-by-Step Implementation

```
from sklearn.decomposition import PCA  
from sklearn.datasets import load_iris
```

```
# Load dataset
```

```
iris = load_iris()  
X = iris.data
```

```
# Apply PCA
```

```
pca = PCA(n_components=2)  
X_pca = pca.fit_transform(X)
```

```
# Plot PCA results
```

```
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=iris.target, cmap='coolwarm')
```

```
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('PCA - Iris Dataset')
plt.show()
```

**Outcome:** Data is visualized in a **lower-dimensional space**.

## 5. t-SNE for High-Dimensional Data Visualization

t-SNE is used for **visualizing high-dimensional datasets in 2D or 3D**.

### Syntax:

```
from sklearn.manifold import TSNE
```

```
tsne = TSNE(n_components=2, perplexity=30, random_state=42)
X_tsne = tsne.fit_transform(X)
```

## Real-Life Example: Visualizing Handwritten Digits

**Problem:** A researcher wants to visualize handwritten digits (MNIST dataset).

### Step-by-Step Implementation

```
from sklearn.datasets import load_digits
from sklearn.manifold import TSNE
import seaborn as sns

# Load dataset
```

```
digits = load_digits()  
X = digits.data  
y = digits.target  
  
# Apply t-SNE  
tsne = TSNE(n_components=2, perplexity=30, random_state=42)  
X_tsne = tsne.fit_transform(X)  
  
# Plot t-SNE results  
plt.figure(figsize=(8, 6))  
sns.scatterplot(x=X_tsne[:, 0], y=X_tsne[:, 1], hue=y, palette='tab10')  
plt.title("t-SNE Visualization of Handwritten Digits")  
plt.show()
```

**Outcome:** Digits are clustered into groups based on similarity.

## Conclusion

- ◆ **K-Means, Hierarchical, and DBSCAN** help in **clustering** data.
- ◆ **PCA and t-SNE** help in **dimensionality reduction and visualization**.
- ◆ **Real-world applications** include customer segmentation, fraud detection, and feature selection.

## Mini Project 1: Customer Segmentation for an E-Commerce Platform

### Objective:

Segment customers based on purchasing behavior using **K-Means, Hierarchical Clustering, and DBSCAN** to identify high-value, medium-value, and low-value customers.

### Step 1: Import Required Libraries

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans, DBSCAN, AgglomerativeClustering
import scipy.cluster.hierarchy as sch
```

### Step 2: Load and Explore the Dataset

```
# Sample dataset representing customers' annual income and spending score
data = pd.DataFrame({
    'Customer ID': range(1, 21),
    'Annual Income (k$)': [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 55, 60, 65, 70, 75,
                           80, 85, 90, 95, 100],
    'Spending Score (1-100)': [39, 81, 6, 77, 40, 76, 6, 94, 3, 72, 20, 80, 13, 85, 5, 90,
                               8, 95, 12, 100]
})

# Extracting relevant features
X = data.iloc[:, 1:].values
```

```
# Display first 5 rows
print(data.head())
```

### Step 3: Standardize Data

```
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

### Step 4: Apply K-Means Clustering

```
# Determine the optimal number of clusters using Elbow Method
wcss = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters=i, random_state=42)
    kmeans.fit(X_scaled)
    wcss.append(kmeans.inertia_)
```

```
# Plot the Elbow graph
plt.figure(figsize=(6, 4))
plt.plot(range(1, 11), wcss, marker='o', linestyle='--')
plt.xlabel('Number of Clusters')
plt.ylabel('WCSS')
plt.title('Elbow Method for Optimal k')
plt.show()
```

**Choose the optimal number of clusters where the "elbow" occurs. Let's say k=3.**

```
# Apply K-Means with optimal k
kmeans = KMeans(n_clusters=3, random_state=42)
```

1080

**Private & Confidential : Vetri Technology Solutions**

```

data['KMeans_Cluster'] = kmeans.fit_predict(X_scaled)

# Plot Clusters
plt.scatter(X[:, 0], X[:, 1], c=data['KMeans_Cluster'], cmap='viridis')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
            s=300, c='red', marker='X', label='Centroids')
plt.xlabel('Annual Income (k$)')
plt.ylabel('Spending Score')
plt.legend()
plt.title('K-Means Clustering')
plt.show()

```

### Step 5: Apply Hierarchical Clustering

```

# Create Dendrogram
plt.figure(figsize=(6, 4))
sch.dendrogram(sch.linkage(X_scaled, method='ward'))
plt.title('Dendrogram for Hierarchical Clustering')
plt.xlabel('Customers')
plt.ylabel('Distance')
plt.show()

# Apply Hierarchical Clustering
hc = AgglomerativeClustering(n_clusters=3)
data['Hierarchical_Cluster'] = hc.fit_predict(X_scaled)

# Plot Clusters
plt.scatter(X[:, 0], X[:, 1], c=data['Hierarchical_Cluster'], cmap='coolwarm')
plt.xlabel('Annual Income (k$)')
plt.ylabel('Spending Score')

```

```
plt.title('Hierarchical Clustering')
plt.show()
```

### Step 6: Apply DBSCAN

```
dbscan = DBSCAN(eps=0.5, min_samples=3)
data['DBSCAN_Cluster'] = dbscan.fit_predict(X_scaled)

# Plot Clusters
plt.scatter(X[:, 0], X[:, 1], c=data['DBSCAN_Cluster'], cmap='rainbow')
plt.xlabel('Annual Income (k$)')
plt.ylabel('Spending Score')
plt.title('DBSCAN Clustering')
plt.show()
```

**Outcome:** Customers are divided into different segments based on spending behavior.

## Mini Project 2: High-Dimensional Data Visualization of Handwritten Digits

### Objective:

Use **PCA** and **t-SNE** to reduce dimensions of the **MNIST dataset** (handwritten digits) and visualize the clusters.

**Step 1: Import Required Libraries**

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
from sklearn.datasets import load_digits
from sklearn.preprocessing import StandardScaler
```

**Step 2: Load and Explore the Dataset**

```
# Load MNIST dataset (Handwritten digits 0-9)
digits = load_digits()
X = digits.data # Features
y = digits.target # Labels

# Display dataset shape
print("Dataset Shape:", X.shape) # (1797, 64)
```

Each image has **64 features** (8x8 pixel grayscale image).

**Step 3: Standardize Data**

```
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

### Step 4: Apply PCA for Dimensionality Reduction

```
# Reduce to 2 Principal Components
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

# Plot PCA results
plt.figure(figsize=(8, 6))
sns.scatterplot(x=X_pca[:, 0], y=X_pca[:, 1], hue=y, palette='tab10', alpha=0.7)
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('PCA - Handwritten Digits')
plt.show()
```

**Outcome:** The dataset is reduced from **64 dimensions to 2** for visualization.

### Step 5: Apply t-SNE for High-Dimensional Data Visualization

```
tsne = TSNE(n_components=2, perplexity=30, random_state=42)
X_tsne = tsne.fit_transform(X_scaled)

# Plot t-SNE results
plt.figure(figsize=(8, 6))
sns.scatterplot(x=X_tsne[:, 0], y=X_tsne[:, 1], hue=y, palette='tab10', alpha=0.7)
plt.xlabel('t-SNE Component 1')
plt.ylabel('t-SNE Component 2')
plt.title('t-SNE - Handwritten Digits')
plt.show()
```

**Outcome:**

- PCA provides a **linear** dimensionality reduction.

- t-SNE captures **non-linear relationships** and shows clear clustering of digits.

## Final Thoughts

- ◆ **Mini Project 1:** Used K-Means, Hierarchical, and DBSCAN to segment customers based on spending patterns.
- ◆ **Mini Project 2:** Used PCA and t-SNE to visualize high-dimensional handwritten digit data.

Would you like any modifications or additional projects? 

## Mini Project: Movie Recommendation System Using Clustering & Dimensionality Reduction

### Objective:

Cluster movies based on user ratings and visualize relationships between genres using **K-Means, Hierarchical Clustering, DBSCAN, PCA, and t-SNE**.

### ◆ Tasks Breakdown (13 Tasks)

1. **Data Collection:** Download a movie dataset containing user ratings, genres, and other metadata.
2. **Data Preprocessing:** Handle missing values, remove duplicates, and standardize data.
3. **Feature Selection:** Choose relevant features such as user ratings, genres, and popularity score.
4. **Data Normalization:** Use StandardScaler to normalize the dataset before applying clustering.

5. **Applying K-Means Clustering:** Determine the optimal number of clusters using the **Elbow Method** and implement K-Means.
6. **Visualizing K-Means Clusters:** Plot clusters based on selected movie features.
7. **Applying Hierarchical Clustering:** Create a **dendrogram** and group movies into meaningful clusters.
8. **Applying DBSCAN Clustering:** Use **DBSCAN** to detect noise and separate core clusters.
9. **Dimensionality Reduction with PCA:** Reduce the high-dimensional movie features into **two principal components** and visualize the data.
10. **Applying t-SNE for Non-Linear Relationships:** Use **t-SNE** to reveal hidden structures in movie preferences.
11. **Evaluating Clustering Performance:** Compare clustering results using silhouette scores and inertia.
12. **Movie Recommendation Based on Clusters:** Suggest movies to users based on the cluster they belong to.
13. **Final Report & Visualization:** Present the findings with graphs, cluster insights, and recommendations.

#### **Outcome:**

- Movies are grouped into meaningful clusters based on user preferences.
- A recommendation system is built using clustering models.
- High-dimensional movie data is visualized effectively.

## Mini Project 1: Social Media User Behavior Analysis

### Objective:

Analyze user behavior on a social media platform and segment users into different categories based on engagement, post frequency, and interaction patterns.

### Key Techniques:

- Use **K-Means, Hierarchical Clustering, and DBSCAN** to group users into different engagement levels.
- Apply **PCA** to reduce dimensionality and improve clustering performance.
- Visualize user behavior patterns using **t-SNE**.

### Expected Outcome:

- Identify user groups such as active users, passive users, and influencers.
- Provide insights for targeted advertising and content recommendations.

## Mini Project 2: Image Compression Using PCA & Clustering

### Objective:

Use **Principal Component Analysis (PCA)** and clustering methods to compress images while preserving essential details.

### Key Techniques:

- Convert image pixels into numerical features and apply **PCA** to reduce dimensionality.
- Use **K-Means Clustering** to segment image regions and improve compression.

- Compare compression quality with different numbers of **principal components** and clusters.
- Visualize reconstructed images and analyze quality loss.

**Expected Outcome:**

- Reduce image storage space while maintaining visual quality.
- Implement a scalable approach for compressing large datasets of images.

## Day 74

### Model Deployment & Real-World Applications: A Complete Guide

Model deployment is the process of making a trained machine learning (ML) model available for real-world use. It allows businesses and applications to make predictions in real-time using previously trained models. This guide covers:

- ✓ Saving & Loading Models (`joblib`, `pickle`)
- ✓ Deploying ML Models with Flask/FastAPI
- ✓ Introduction to ML Pipelines (`Pipeline()`)
- ✓ End-to-End ML Project using a Real-World Dataset

## 1. Saving & Loading Models (joblib, pickle)

### Definition

After training an ML model, it's important to save it so that we can reuse it later without retraining. We use **pickle** and **joblib** for this purpose.

### Syntax

#### Using pickle to save and load models

```
import pickle
```

```
# Save the model
with open('model.pkl', 'wb') as file:
    pickle.dump(model, file)
```

```
# Load the model
with open('model.pkl', 'rb') as file:
    loaded_model = pickle.load(file)
```

#### Using joblib (better for large models)

```
from joblib import dump, load
```

```
# Save the model
dump(model, 'model.joblib')
```

```
# Load the model
loaded_model = load('model.joblib')
```

## Real-Life Example (Step-by-Step)

**Scenario:** You have trained a house price prediction model, and you want to save and reload it later.

### Step 1: Train a Simple Model

```
from sklearn.linear_model import LinearRegression  
import numpy as np  
  
# Sample data  
X = np.array([[1000], [1500], [2000], [2500], [3000]]) # Square feet  
y = np.array([100000, 150000, 200000, 250000, 300000]) # Price in $  
  
# Train the model  
model = LinearRegression()  
model.fit(X, y)
```

### Step 2: Save the Model using joblib

```
from joblib import dump  
  
dump(model, 'house_price_model.joblib')  
print("Model saved successfully!")
```

### Step 3: Load the Model & Make Predictions

```
from joblib import load  
  
# Load the model
```

```

loaded_model = load('house_price_model.joblib')

# Make a prediction
sqft = np.array([[1800]]) # House size in square feet
predicted_price = loaded_model.predict(sqft)
print(f"Predicted Price: ${predicted_price[0]:,.2f}")

```

## 2. Deploying ML Models with Flask/FastAPI

### Definition

Flask and FastAPI are Python web frameworks that allow us to deploy machine learning models as web APIs.

### Flask vs FastAPI

Feature	Flask	FastAPI
Speed	Slower	Faster (Async)
Performance	Good	Better
Syntax	Traditional	Modern (Type Hints)

### Real-Life Example: Deploying a Model with Flask

**Scenario:** You want to deploy the house price prediction model as an API so users can send requests and get price predictions.

#### Step 1: Install Flask

```
pip install flask
```

**Step 2: Create app.py**

```
from flask import Flask, request, jsonify
from joblib import load
import numpy as np

app = Flask(__name__)

# Load the saved model
model = load('house_price_model.joblib')

@app.route('/predict', methods=['POST'])
def predict():
    data = request.get_json() # Receive JSON input
    sqft = np.array([[data['sqft']]]) # Extract input value
    prediction = model.predict(sqft)[0] # Make prediction
    return jsonify({'predicted_price': round(prediction, 2)}) # Send response

if __name__ == '__main__':
    app.run(debug=True)
```

**Step 3: Run the API**

python app.py

- The API will be available at: <http://127.0.0.1:5000/predict>

**Step 4: Test API with Postman or Curl**

Send a **POST** request with JSON input:

```
{  
  "sqft": 1800  
}
```

Response:

```
{  
  "predicted_price": 180000.0  
}
```

### 3. Introduction to ML Pipelines (Pipeline())

#### **Definition**

A pipeline automates the entire machine learning workflow, including preprocessing, training, and prediction.

#### **Syntax**

```
from sklearn.pipeline import Pipeline  
from sklearn.preprocessing import StandardScaler  
from sklearn.linear_model import LinearRegression  
  
# Create a pipeline  
pipe = Pipeline([  
    ('scaler', StandardScaler()), # Step 1: Scaling  
    ('model', LinearRegression()) # Step 2: Model  
])
```

```
# Train the pipeline  
pipe.fit(X, y)  
  
# Predict  
prediction = pipe.predict([[1800]])
```

### Real-Life Example

**Scenario:** You need to preprocess house data before training the model.

#### Step 1: Create a Pipeline

```
from sklearn.pipeline import Pipeline  
from sklearn.preprocessing import StandardScaler  
from sklearn.linear_model import LinearRegression  
import numpy as np  
  
# Sample data  
X = np.array([[1000], [1500], [2000], [2500], [3000]])  
y = np.array([100000, 150000, 200000, 250000, 300000])  
  
# Define pipeline  
pipe = Pipeline([  
    ('scaler', StandardScaler()),  
    ('model', LinearRegression())  
])  
  
# Train the pipeline  
pipe.fit(X, y)  
  
# Make a prediction
```

```
prediction = pipe.predict([[1800]])
print(f"Predicted Price: ${prediction[0]:.2f}")
```

## 4. End-to-End ML Project (Real-World Dataset)

### Definition

An end-to-end ML project involves **data collection, preprocessing, model training, evaluation, and deployment.**

### Real-Life Example

**Scenario:** Build an end-to-end **customer churn prediction system** using ML.

#### Step 1: Load Dataset

```
import pandas as pd

df = pd.read_csv('customer_churn.csv')
print(df.head())
```

#### Step 2: Preprocess Data

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Separate features and target
X = df.drop(columns=['Churn'])
y = df['Churn']
```

```
# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Scale the data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

### Step 3: Train Model

```
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier()
model.fit(X_train_scaled, y_train)
```

### Step 4: Evaluate Model

```
from sklearn.metrics import accuracy_score

y_pred = model.predict(X_test_scaled)
accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy: {accuracy:.2f}")
```

### Step 5: Save & Deploy Model (Same as Flask API Steps)

## Mini Project 1: Movie Review Sentiment Analysis Deployment using Flask

### Objective:

Develop a **sentiment analysis model** for **movie reviews**, save the trained model, and deploy it using **Flask**. Users can input a review, and the model will predict if it is **positive or negative**.

### Step 1: Install Required Libraries

```
pip install numpy pandas scikit-learn flask joblib
```

### Step 2: Prepare the Dataset

We use a sample dataset of **movie reviews** labeled as 1 (positive) or 0 (negative).

```
import pandas as pd
```

```
# Creating a simple dataset
data = {
    'review': [
        "This movie was fantastic! I loved it.",
        "Absolutely terrible. Waste of time.",
        "The plot was interesting and well-executed.",
        "Horrible acting and bad direction.",
        "Great cinematography and amazing soundtrack."
    ],
    'sentiment': [1, 0, 1, 0, 1] # 1 = Positive, 0 = Negative
}
df = pd.DataFrame(data)
```

**Step 3: Preprocess Data & Train the Model**

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline

# Splitting input (X) and target (y)
X = df['review']
y = df['sentiment']

# Create a pipeline with text vectorization and model
pipeline = Pipeline([
    ('tfidf', TfidfVectorizer()), # Convert text to numeric features
    ('model', MultinomialNB()) # Train Naive Bayes classifier
])

# Train the model
pipeline.fit(X, y)
```

**Step 4: Save & Load the Model**

```
from joblib import dump, load

# Save the trained model
dump(pipeline, 'sentiment_model.joblib')

# Load and test the model
loaded_model = load('sentiment_model.joblib')
print(loaded_model.predict(["I hated this movie."])) # Output: [0] (Negative)
```

## Step 5: Create a Flask API

### Create a file app.py

```
from flask import Flask, request, jsonify
from joblib import load

app = Flask(__name__)

# Load saved model
model = load('sentiment_model.joblib')

@app.route('/predict', methods=['POST'])
def predict():
    data = request.get_json()
    review_text = [data['review']]
    prediction = model.predict(review_text)[0]
    sentiment = "Positive" if prediction == 1 else "Negative"
    return jsonify({'sentiment': sentiment})

if __name__ == '__main__':
    app.run(debug=True)
```

## Step 6: Run the Flask API

python app.py

- The API will be available at <http://127.0.0.1:5000/predict>

### Step 7: Test API Using Postman or Curl

Send a **POST request** with JSON:

```
{  
  "review": "The movie was boring and predictable."  
}
```

Expected Response:

```
{  
  "sentiment": "Negative"  
}
```

✓ Congratulations! Your Sentiment Analysis Model is Live. 🎉

## Mini Project 2: Loan Approval Prediction with FastAPI

**Objective:**

Develop a **loan approval prediction model**, use **ML Pipelines** for automation, and deploy the model using **FastAPI**.

### Step 1: Install Required Libraries

```
pip install numpy pandas scikit-learn fastapi uvicorn joblib
```

## Step 2: Load & Preprocess Data

We use a **loan approval dataset**, where 1 means the loan is approved, and 0 means it is rejected.

```
import pandas as pd

# Sample dataset
data = {
    'income': [30000, 50000, 70000, 90000, 110000],
    'loan_amount': [5000, 10000, 15000, 20000, 25000],
    'credit_score': [600, 650, 700, 750, 800],
    'loan_approved': [0, 1, 1, 1, 1] # 1 = Approved, 0 = Rejected
}

df = pd.DataFrame(data)

# Splitting into features and target
X = df.drop(columns=['loan_approved'])
y = df['loan_approved']
```

## Step 3: Build a Machine Learning Pipeline

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

```
# Create an ML pipeline
pipeline = Pipeline([
    ('scaler', StandardScaler()), # Feature scaling
    ('model', RandomForestClassifier(n_estimators=100, random_state=42))
])

# Train the pipeline
pipeline.fit(X_train, y_train)
```

#### Step 4: Save & Load the Model

```
from joblib import dump, load

# Save the pipeline
dump(pipeline, 'loan_approval_model.joblib')

# Load the pipeline
loaded_pipeline = load('loan_approval_model.joblib')

# Test prediction
test_data = [[60000, 12000, 680]] # Example input
loan_approval = loaded_pipeline.predict(test_data)
print(f"Loan Approval: {'Approved' if loan_approval[0] == 1 else 'Rejected'}")
```

## Step 5: Deploy Model with FastAPI

### Create a file loan\_api.py

```
from fastapi import FastAPI
from joblib import load
import numpy as np

app = FastAPI()

# Load model
model = load('loan_approval_model.joblib')

@app.post("/predict")
def predict_loan(income: int, loan_amount: int, credit_score: int):
    input_data = np.array([[income, loan_amount, credit_score]])
    prediction = model.predict(input_data)[0]
    return {"loan_approval": "Approved" if prediction == 1 else "Rejected"}
```

## Step 6: Run the FastAPI App

```
uvicorn loan_api:app --reload
```

- The API will be available at <http://127.0.0.1:8000/predict>

## Step 7: Test API Using Postman or Browser

Send a **POST request**:

[http://127.0.0.1:8000/predict?income=70000&loan\\_amount=15000&credit\\_score=700](http://127.0.0.1:8000/predict?income=70000&loan_amount=15000&credit_score=700)

Response:

```
{  
    "loan_approval": "Approved"  
}
```

Congratulations! Your Loan Approval Prediction API is Live. 

## Real-Life Mini Project: Fraud Detection System Deployment

Objective:

Develop a **Fraud Detection Model** that predicts whether a transaction is **fraudulent or legitimate**, optimize it with **pipelines**, save & load the trained model, and deploy it using **FastAPI**.

### 13 Tasks for This Mini Project

1. **Dataset Collection:** Find and download a real-world fraud detection dataset (e.g., Kaggle credit card fraud dataset).
2. **Data Preprocessing:** Handle missing values, remove duplicates, and normalize numeric features.
3. **Feature Engineering:** Select important features using feature importance analysis.
4. **Splitting Dataset:** Split the dataset into training and testing sets.

5. **Model Selection:** Train different models (Logistic Regression, Random Forest, XGBoost) to find the best one.
6. **Hyperparameter Tuning:** Use **GridSearchCV** or **RandomizedSearchCV** to optimize model parameters.
7. **Building an ML Pipeline:** Create a **Pipeline()** that automates preprocessing, feature selection, and training.
8. **Evaluating the Model:** Use accuracy, precision, recall, and F1-score to assess the model's performance.
9. **Saving & Loading Model:** Save the trained model using **joblib/pickle** and test loading it.
10. **Deploying Model with FastAPI:** Build an **API endpoint** that receives transaction data and returns fraud prediction.
11. **Testing API Locally:** Use **Postman** or **cURL** to send sample transaction data and verify predictions.
12. **Containerizing with Docker (Optional):** Create a **Docker container** for easy deployment.
13. **Deploying on Cloud:** Deploy the model on **AWS, Render, or Heroku** for real-world usage.

### Expected Outcomes

- ✓ A trained **Fraud Detection Model** that can identify fraudulent transactions.
- ✓ A **FastAPI-based API** where users send transaction details and get fraud predictions.
- ✓ A **fully optimized ML pipeline with hyperparameter tuning & feature selection.**
- ✓ The model can be **loaded, tested, and deployed** for real-world applications.

## Updated Two Real-Life Mini Project Requirements

### Mini Project 1: Fake News Detection System

**Objective:** Develop a **Machine Learning model to detect fake news** based on article text.

#### Key Requirements:

- Use a **real-world dataset** containing fake and real news articles.
- Perform **EDA (Exploratory Data Analysis)** to clean and preprocess text data.
- Apply **TF-IDF vectorization or word embeddings** to convert text into numerical format.
- Train **Logistic Regression, Random Forest, and Support Vector Machine (SVM) classifiers**.
- Perform **hyperparameter tuning using GridSearchCV** to optimize model performance.
- Evaluate models using **accuracy, precision, recall, and F1-score**.
- Save the best model using **pickle or joblib**.
- Deploy a **Flask or FastAPI API** where users can input news articles and get predictions.
- Deploy the API on **AWS, Heroku, or Render**, and build a **basic web interface**.

### Mini Project 2: Credit Card Fraud Detection System

**Objective:** Build a **Credit Card Fraud Detection Model** to identify fraudulent transactions.

### Key Requirements:

- Use a **real-world financial dataset** containing transaction details labeled as fraud or non-fraud.
- Perform **EDA, feature selection, and feature engineering** to improve model accuracy.
- Train **Random Forest, Decision Tree, and K-Nearest Neighbors (KNN) models**.
- Handle **imbalanced data using SMOTE (Synthetic Minority Over-sampling Technique)**.
- Perform **cross-validation and hyperparameter tuning (GridSearchCV, RandomizedSearchCV)**.
- Evaluate model performance using **precision, recall, F1-score, and confusion matrix**.
- Save the trained model using **joblib or pickle**.
- Deploy a **FastAPI API** where users can input transaction details and get fraud detection results.
- Deploy on **cloud platforms** and integrate a **dashboard for real-time monitoring**.

# Flask

## Day 75

### Introduction to Flask & Setup (Step-by-Step Guide)

#### What is Flask?

Flask is a **lightweight and flexible web framework** for Python used to build web applications. It is simple to use and allows for quick development of web apps and APIs.

Flask is called a "**micro-framework**" because it does not include built-in features like an ORM (Object Relational Mapper) or authentication system, which makes it more flexible compared to Django.

#### Why Use Flask Over Django?

Feature	Flask	Django
<b>Flexibility</b>	More flexible, minimal setup	Comes with built-in features
<b>Learning Curve</b>	Easier to learn	More complex due to built-in features
<b>Project Type</b>	Best for small apps, APIs	Best for large, full-stack apps
<b>Speed</b>	Faster for small projects	Slightly slower due to more components

Flask is great if you need **control** over your app structure and want a lightweight framework. Django is better for **large applications** that need built-in features like authentication, admin panel, and ORM.

## Step 1: Installing Flask

Before starting, ensure you have Python installed (Python 3.x recommended).

### Install Flask using pip

```
pip install flask
```

You can verify the installation using:

```
python -m flask --version
```

## Step 2: Creating Your First Flask App (app.py)

Now, let's create a **basic Flask application**.

### Create a Python file named app.py

```
from flask import Flask # Import Flask

app = Flask(__name__) # Initialize the Flask app

@app.route('/') # Define a route for the homepage
def home():
    return "Hello, Flask! This is your first web app."

if __name__ == '__main__':
    app.run(debug=True) # Run the Flask development server
```

### Step 3: Running the Flask Development Server

**Open your terminal and run:**

```
python app.py
```

#### Output in Terminal

\* Running on <http://127.0.0.1:5000/>

This means your Flask app is running on **port 5000**. Open your browser and visit:

<http://127.0.0.1:5000/>

**You should see:**

Hello, Flask! This is your first web app.

### Step 4: Understanding @app.route() in Flask

The `@app.route()` decorator maps a URL to a function. It tells Flask which function to execute when a user visits a specific URL.

#### Example: Adding More Routes

```
from flask import Flask
```

```
app = Flask(__name__)
```

```
@app.route('/') # Homepage route
```

```
def home():
```

```
    return "Welcome to the Homepage!"
```

```
@app.route('/about') # About page route
def about():
    return "This is the About Page."

if __name__ == '__main__':
    app.run(debug=True)
```

### Running the App

- **Homepage:** <http://127.0.0.1:5000/> → Displays "**Welcome to the Homepage!**"
- **About Page:** <http://127.0.0.1:5000/about> → Displays "**This is the About Page.**"

### Step 5: Returning Simple HTML Responses

Flask can return **HTML code** instead of plain text.

#### Example: Returning HTML Content

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return "<h1>Welcome to Flask</h1><p>This is a simple HTML response.</p>"

@app.route('/contact')
def contact():
    return """"
```

```

<h1>Contact Us</h1>
<p>Email: support@example.com</p>
<p>Phone: +123 456 7890</p>
"""

if __name__ == '__main__':
    app.run(debug=True)

```

### Open in Browser:

- **Homepage:** <http://127.0.0.1:5000/>
- **Contact Page:** <http://127.0.0.1:5000/contact>

**You should see formatted HTML content on both pages!**

### Summary

Step	Topic	Description
1	Install Flask	pip install flask
2	Create Flask App	Write app.py and define routes
3	Run Flask Server	python app.py, visit <a href="http://127.0.0.1:5000/">http://127.0.0.1:5000/</a>
4	Using @app.route()	Map URLs to functions
5	Return HTML Responses	Return formatted HTML instead of plain text

## Mini Project 1: Simple Personal Website using Flask

### Overview:

In this project, we will build a **simple personal website** using Flask. The website will have a **Home page, About page, and Contact page** with simple HTML responses.

### Step 1: Install Flask

First, install Flask if you haven't already. Open your terminal or command prompt and run:

```
pip install flask
```

### Step 2: Create the Project Folder

Create a new folder for your project and navigate to it:

```
mkdir flask_website  
cd flask_website
```

### Step 3: Create app.py (Main Flask File)

Inside your flask\_website folder, create a new file called **app.py** and add the following code:

```
from flask import Flask # Import Flask  
  
app = Flask(__name__) # Initialize the Flask app  
  
@app.route('/') # Home route
```

```
def home():
    return "<h1>Welcome to My Website</h1><p>This is the Home Page.</p>"

@app.route('/about') # About page route
def about():
    return "<h1>About Me</h1><p>I'm a Python developer learning Flask!</p>"

@app.route('/contact') # Contact page route
def contact():
    return "<h1>Contact</h1><p>Email: example@example.com</p><p>Phone:<br>+123456789</p>"

if __name__ == '__main__':
    app.run(debug=True) # Run Flask server
```

#### **Step 4: Run the Flask Development Server**

Now, open your terminal, navigate to your project folder, and run:

```
python app.py
```

You should see output like this:

```
* Running on http://127.0.0.1:5000/
```

#### **Step 5: Test in Your Browser**

- Open <http://127.0.0.1:5000/> → Home Page
- Open <http://127.0.0.1:5000/about> → About Page
- Open <http://127.0.0.1:5000/contact> → Contact Page

## Mini Project 2: Simple To-Do List App

### Overview:

In this project, we will build a **simple To-Do List app** that allows users to view and add tasks using Flask.

### Step 1: Install Flask

Ensure Flask is installed:

```
pip install flask
```

### Step 2: Create the Project Folder

```
mkdir flask_todo  
cd flask_todo
```

### Step 3: Create app.py (Main Flask File)

Inside your flask\_todo folder, create a new file **app.py** and add the following code:

```
from flask import Flask, request  
  
app = Flask(__name__) # Initialize Flask app  
  
tasks = [] # List to store tasks  
  
@app.route('/') # Home route  
def home():
```

```

return f"<h1>To-Do List</h1><ul>{".join(f'<li>{task}</li>' for task in
tasks)}</ul><br><a href='/add'>Add Task</a>"

@app.route('/add', methods=['GET', 'POST']) # Add task route
def add_task():
    if request.method == 'POST':
        task = request.form.get('task')
        if task:
            tasks.append(task)
        return home()
    return """
<h1>Add a Task</h1>
<form method="post">
    Task: <input type="text" name="task">
    <input type="submit" value="Add">
</form>
<br><a href='/'>Go Back</a>
"""

if __name__ == '__main__':
    app.run(debug=True) # Run Flask server

```

#### **Step 4: Run the Flask Server**

Run the app using:

python app.py

You will see:

\* Running on <http://127.0.0.1:5000/>

## Step 5: Test the App in Your Browser

1. **Go to Home Page:** <http://127.0.0.1:5000/>
  - a. You will see an empty To-Do List.
  - b. Click "Add Task" to add a new task.
2. **Add a Task:** <http://127.0.0.1:5000/add>
  - a. Enter a task in the form and submit.
  - b. It will be added to your To-Do list.

**Congratulations! You have built a basic To-Do List app using Flask!** 🎉

## Summary

Project	Features Covered
Personal Website	Basic Flask app, routes ( <code>@app.route()</code> ), HTML responses
To-Do List App	Handling user input, displaying lists, simple form submission

## Tasks :

### 1. What is Flask?

- Research and write a short explanation of Flask and why it is used in web development.
- Compare Flask with Django and explain when to choose Flask over Django.

### 2. Install Flask on Your System

- Check if Python is installed on your system.
- Install Flask using pip.
- Verify the installation.

### 3. Create a Basic Flask App (app.py)

1117

Private & Confidential : Vetri Technology Solutions

- Set up a simple **Flask app** with just the **Flask object initialization**.
- Run the Flask app and check if it works.

#### 4. Run the Flask Development Server

- Use the correct command to run your Flask app.
- Identify which port the Flask development server runs on by default.

#### 5. Understanding Flask Routes (@app.route())

- Create two different routes (/ and /about).
- Explain how Flask maps URLs to functions.

#### 6. Returning a Simple HTML Response

- Modify your Flask app to return a **simple HTML response** instead of plain text.

#### 7. Add Multiple Routes to Your Flask App

- Add an /about and /contact route to your app.
- Ensure that each route returns different content.

#### 8. Use Variables in Routes

- Create a route that takes a **user's name** as a variable (e.g., /hello/<name>).
- Return a dynamic message using that name.

#### 9. Handle Different HTTP Methods (GET and POST)

- Create a new route that **accepts both GET and POST requests**.
- Explain how Flask handles HTTP methods.

#### 10. Use Flask's Debug Mode

- Enable Flask's debug mode and observe what happens when you make a syntax error.
- Explain why debug mode is useful during development.

### 11. Create a Simple Navigation System

- Modify your Flask app to include **links between pages** for easy navigation.

### 12. Return an HTML File Instead of a String

- Store an HTML file (index.html) inside a folder and return it using Flask.
- Explain why this is useful in larger applications.

### 13. Deploy a Basic Flask App on a Local Server

- Learn and execute the steps to **deploy your Flask app locally**.
- Check if the server is running in the background.

## Mini Project 1: "Personalized Greeting Flask App"

**Objective:** Create a Flask app that takes a user's name as input via a URL and displays a personalized greeting.

Project Steps:

1. **Install Flask** using pip install flask.
2. **Create a Flask app (app.py)** with a basic route.
3. **Use @app.route() to create a dynamic route** that accepts a name parameter (e.g., /hello/<name>).
4. **Return a personalized greeting message** using HTML.

5. Run the Flask development server and test the route in a web browser.

**Example Output:**

- Visiting /hello/John should display: "Hello, John! Welcome to Flask!"

### **Mini Project 2: "Simple Web Page with Multiple Routes"**

**Objective:** Build a Flask app with multiple routes that serve different pages, like a homepage and an about page.

Project Steps:

1. Install Flask if not already installed.

2. Create a Flask app (`app.py`) with at least two routes:

- / → Home Page
- /about → About Page

3. Use `@app.route()` to define the routes and return simple HTML responses.

4. Run the Flask development server and test both routes in a web browser.

✓ **Example Output:**

- Visiting / should display: "Welcome to My Flask Website!"
- Visiting /about should display: "This is a simple Flask web app created by [Your Name]."

# Day 76

## Flask Routing & Request Handling - Step-by-Step Explanation with Examples

Flask provides powerful routing and request handling features that allow you to create dynamic web applications. Let's go through each concept with a **definition, syntax, and easy step-by-step examples**.

### 1. URL Routing & Dynamic Parameters (/<name>)

#### Definition:

Routing in Flask means **mapping a URL to a specific function** in your Python application. Dynamic parameters allow us to **pass values** inside the URL.

#### Syntax:

```
@app.route('/hello/<name>')
def greet(name):
    return f"Hello, {name}!"
```

#### Step-by-Step Implementation:

1. **Install Flask** (if not installed): pip install flask
2. **Create app.py and define a route with a dynamic parameter:** from flask import Flask

```

app = Flask(__name__)

@app.route('/hello/<name>')
def greet(name):
    return f"Hello, {name}!"

if __name__ == "__main__":
    app.run(debug=True)

```

**3. Run the Flask app:** python app.py

**4. Open in browser:**

- a. Visit <http://127.0.0.1:5000/hello/John>
- b. Output: "Hello, John!"

## 2. Handling GET & POST Requests

### Definition:

Flask allows handling different request methods like GET and POST.

- GET → Used for retrieving data.
- POST → Used for submitting data.

### Syntax:

```

@app.route('/submit', methods=['GET', 'POST'])
def handle_form():
    if request.method == 'POST':
        return "Form Submitted!"

```

```
return "Please Submit the Form"
```

### **Step-by-Step Implementation:**

- 1. Modify app.py to handle GET and POST requests:** from flask import Flask, request

```
app = Flask(__name__)

@app.route('/submit', methods=['GET', 'POST'])
def handle_form():
    if request.method == 'POST':
        return "Form Submitted!"
    return "Please Submit the Form"

if __name__ == "__main__":
    app.run(debug=True)
```

- 2. Run the Flask app:** python app.py

- 3. Test in browser:**

- a. Visit <http://127.0.0.1:5000/submit>** → It will show "Please Submit the Form"
- b. Use a tool like Postman or an HTML form to send a POST request,** it will return "Form Submitted!"

### 3. Using request.args for Query Parameters

#### Definition:

Query parameters are used to **pass data in the URL**. We access them using `request.args`.

#### Syntax:

```
@app.route('/search')
def search():
    name = request.args.get('name')
    return f"Searching for {name}"
```

#### Step-by-Step Implementation:

##### 1. Add this route to `app.py`: from flask import Flask, request

```
app = Flask(__name__)

@app.route('/search')
def search():
    name = request.args.get('name', 'Guest')
    return f"Searching for {name}"

if __name__ == "__main__":
    app.run(debug=True)
```

##### 2. Run Flask and open in browser:

- a. Visit <http://127.0.0.1:5000/search?name=Python>
- b. Output: "Searching for Python"

#### 4. Handling Form Data with `request.form`

##### Definition:

`request.form` is used to **handle form data submitted via POST requests**.

##### Syntax:

```
@app.route('/submit', methods=['POST'])
def handle_form():
    name = request.form['name']
    return f"Hello, {name}!"
```

##### Step-by-Step Implementation:

1. **Create an HTML form (templates/form.html):** <form action="/submit" method="POST">  
    <input type="text" name="name" placeholder="Enter your name">  
    <button type="submit">Submit</button>  
  </form>

2. **Modify app.py to handle form submission:** from flask import Flask, request, render\_template

```
app = Flask(__name__)

@app.route('/')
def form():
    return render_template('form.html')

@app.route('/submit', methods=['POST'])
```

```

def handle_form():
    name = request.form['name']
    return f"Hello, {name}!"

if __name__ == "__main__":
    app.run(debug=True)

```

### 3. Run Flask and test:

- Open <http://127.0.0.1:5000/>
- Enter a name and click submit
- It will display "Hello, <name>!"

## 5. Redirects & URL Building with url\_for()

### Definition:

- redirect() is used to **redirect users** to another route.
- url\_for() dynamically generates URLs.

### Syntax:

```

@app.route('/login')
def login():
    return redirect(url_for('home'))

```

### Step-by-Step Implementation:

- Modify app.py to include redirects:** from flask import Flask, redirect, url\_for

```

app = Flask(__name__)

@app.route('/')
def home():
    return "Welcome to the Home Page!"

@app.route('/login')
def login():
    return redirect(url_for('home'))

if __name__ == "__main__":
    app.run(debug=True)

```

## 2. Run Flask and test:

- Open <http://127.0.0.1:5000/login>
- It will **redirect to / (home page)**

## Summary of Key Concepts

Concept	Description
URL Routing (/<name>)	Handles dynamic parameters in URLs.
Handling GET & POST Requests	Differentiates between retrieving and submitting data.
request.args	Accesses query parameters from the URL.
request.form	Retrieves form data submitted via POST.
redirect(url_for())	Redirects users to a different route dynamically.

## Final Thoughts

This guide covered Flask routing and request handling **step-by-step** with simple examples. You now know how to:

- Create **dynamic routes**
- Handle **GET & POST requests**
- Work with **query parameters & form data**
- Use **redirects & URL building**

## Mini Project 1: User Registration and Greeting App

### Features Covered:

- URL Routing & Dynamic Parameters (/<name>)
- Handling GET & POST Requests
- Using `request.args` for Query Parameters
- Handling Form Data with `request.form`
- Redirects and URL Building (`url_for()`)

### Step-by-Step Implementation

#### Step 1: Install Flask

```
pip install flask
```

## Step 2: Create a Flask App (app.py)

```
from flask import Flask, request, render_template, redirect, url_for

app = Flask(__name__)

# Home Route
@app.route('/')
def home():
    return "Welcome to the User Registration App! <a href='/register'>Register Here</a>"

# Route for Registration Form
@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
        username = request.form['username']
        return redirect(url_for('greet_user', name=username))
    return """
        <form method="POST">
            Name: <input type="text" name="username" required>
            <button type="submit">Register</button>
        </form>
    """

# Route for Greeting User (Dynamic URL Parameter)
@app.route('/greet/<name>')
def greet_user(name):
    return f"Hello, {name}! Welcome to our Flask App!"

if __name__ == "__main__":
```

```
app.run(debug=True)
```

### Step 3: Run Flask

```
python app.py
```

### Step 4: Test the App

- Go to: <http://127.0.0.1:5000/>
- Click Register Here
- Enter a name and submit
- It will redirect to /greet/<name>, displaying a welcome message.

## Mini Project 2: Simple Search Engine

### Features Covered:

- URL Routing (/search)
- Handling GET Requests (request.args)
- Handling Form Data with request.form
- Redirects and URL Building (url\_for())

### Step-by-Step Implementation

#### Step 1: Create Flask App (app.py)

```
from flask import Flask, request, render_template, redirect, url_for
```

```

app = Flask(__name__)

# Home Route with Search Form
@app.route('/')
def home():
    return """
        <h2>Simple Search Engine</h2>
        <form action="/search" method="GET">
            <input type="text" name="query" placeholder="Enter search term"
required>
            <button type="submit">Search</button>
        </form>
    """
    # Search Route (Using request.args)
    @app.route('/search')
    def search():
        query = request.args.get('query')
        if not query:
            return redirect(url_for('home'))

        # Dummy Data for Search
        results = {
            "python": "Python is a programming language.",
            "flask": "Flask is a lightweight web framework.",
            "django": "Django is a full-stack web framework."
        }

        result_text = results.get(query.lower(), "No results found!")
        return f"<h3>Search Results for '{query}'</h3><p>{result_text}</p><br><a href='/'>Go Back</a>"

```

```
if __name__ == "__main__":
    app.run(debug=True)
```

### Step 2: Run Flask

python app.py

### Step 3: Test the App

- Go to: <http://127.0.0.1:5000/>
- Search for "Python", "Flask", or "Django".
- The app will **display search results dynamically**.

### Summary

Feature	Project 1 (User Registration)	Project 2 (Search Engine)
URL Routing & Dynamic Parameters	/greet/<name>	/search?query=<term>
Handling GET & POST Requests	POST for form submission	GET for search queries
request.args (Query Parameters)	✗	✓ (request.args.get('query'))
request.form (Form Handling)	✓ (User Registration Form)	✗ (Only GET request)
Redirects & URL Building (url_for())	✓ Redirect to /greet/<name>	✓ Redirect to /search

## Mini Project: Student Management System (Flask Routing & Request Handling)

This mini-project will help you understand **Flask Routing & Request Handling** by implementing a **Student Management System**. You will complete **13 tasks** related to URL routing, handling GET & POST requests, query parameters, form data, and redirects.

### Tasks

1. Set up Flask
  - a. Install Flask and create a new Flask project (app.py).
2. Create a Home Route (/)
  - a. Display a welcome message with links to view all students and add a student.
3. Define a Route to View All Students (/students)
  - a. Show a list of students stored in a dictionary or list.
4. Implement a Route for Adding a Student (/add\_student)
  - a. Create a form with fields for name and age.
5. Handle Form Submission (POST Request) in /add\_student
  - a. Process form data using request.form and store student details.
6. Use request.args for Filtering Students (/students?age=18)
  - a. Allow filtering students based on age using query parameters.
7. Implement a Dynamic Route for Student Details (/student/<name>)
  - a. Display details of a student when visiting /student/John.
8. Handle Student Not Found Case (/student/<name>)
  - a. If the student doesn't exist, show an error message.
9. Create a Delete Student Route (/delete/<name>)
  - a. Implement a route to remove a student.
10. Use Redirect (url\_for()) After Deleting a Student
  - a. After deletion, redirect to /students.

11. Modify /students to Show a "Delete" Link for Each Student
  - a. Add a delete button next to each student's name.
12. Create a Route (/update/<name>) to Edit Student Details
  - a. Display a form with the current student details.
13. Handle Updating Student Details Using a POST Request (/update/<name>)
  - a. Process the form submission and update the student's information.

### Expected Outcome

- A functional **Flask app** with a **Student Management System**.
- Uses **dynamic routes, GET & POST requests, form handling, query parameters, and redirects**.
- Allows users to **add, view, update, and delete students**.

## Mini Project 1: Blog Post Management System

This project will help you understand **Flask Routing & Request Handling** by implementing a **simple blog system** where users can **view, add, edit, and delete blog posts**.

### Requirements (Without Solutions)

1. Set up Flask and create app.py
  - a. Install Flask and initialize a Flask application.
2. Create a Homepage (/)
  - a. Display a list of all blog posts with a link to add a new post.
3. Implement a Route to View All Posts (/posts)
  - a. Show a list of all blog posts stored in a dictionary or list.
4. Create a Route to Add a New Blog Post (/add\_post)
  - a. Display a form with fields for title and content.

5. Handle Form Submission (POST Request) in /add\_post
  - a. Process form data using request.form and store the post.
6. Use request.args to Filter Posts (/posts?author=John)
  - a. Allow filtering posts based on the author using query parameters.
7. Implement a Dynamic Route to View a Single Post (/post/<title>)
  - a. Display a blog post's title and content when visiting /post/MyFirstPost.
8. Handle Post Not Found Case (/post/<title>)
  - a. If the post doesn't exist, display an error message.
9. Create a Delete Post Route (/delete/<title>)
  - a. Implement a route to remove a post from the list.
10. Use Redirect (url\_for()) After Deleting a Post
  - a. After deletion, redirect to /posts.
11. Modify /posts to Show an "Edit" Link for Each Post
  - a. Add an "Edit" button next to each post title.
12. Create a Route (/edit/<title>) to Update a Blog Post
  - a. Display a form pre-filled with the current post details.
13. Handle Updating a Blog Post Using a POST Request (/edit/<title>)
  - a. Process the form submission and update the post's information.

## Mini Project 2: Employee Management System

This project will help you learn **Flask Routing & Request Handling** by creating an **Employee Management System**, where users can **view, add, update, and delete employee records**.

### Requirements (Without Solutions)

1. Set up Flask and create app.py
  - a. Install Flask and initialize a Flask application.
2. Create a Homepage (/)

- a. Display a welcome message with links to view employees and add an employee.
3. Implement a Route to View All Employees (/employees)
  - a. Show a list of employees stored in a dictionary or list.
4. Create a Route to Add a New Employee (/add\_employee)
  - a. Display a form with fields for name, position, and salary.
5. Handle Form Submission (POST Request) in /add\_employee
  - a. Process form data using request.form and store the employee details.
6. Use request.args to Filter Employees (/employees?position=Manager)
  - a. Allow filtering employees based on job position using query parameters.
7. Implement a Dynamic Route to View an Employee's Details (/employee/<name>)
  - a. Display an employee's details when visiting /employee/JohnDoe.
8. Handle Employee Not Found Case (/employee/<name>)
  - a. If the employee doesn't exist, show an error message.
9. Create a Delete Employee Route (/delete/<name>)
  - a. Implement a route to remove an employee from the list.
10. Use Redirect (url\_for()) After Deleting an Employee
  - a. After deletion, redirect to /employees.
11. Modify /employees to Show an "Update" Link for Each Employee
  - a. Add an "Update" button next to each employee's name.
12. Create a Route (/update/<name>) to Edit Employee Details
  - a. Display a form pre-filled with the employee's details.
13. Handle Updating Employee Details Using a POST Request (/update/<name>)
  - a. Process the form submission and update the employee's information.

## Expected Learning Outcomes

- Understanding of **Flask Routing** with **dynamic parameters**
- Handling **GET & POST** requests effectively
- Using `request.args` for **query parameters**
- Handling **form data** with `request.form`
- Implementing **redirects** and **URL building** using `url_for()`

## Day 77

### Rendering Templates & Jinja2 Templating in Flask

Flask uses the **Jinja2 templating engine** to render dynamic HTML pages. It allows us to **embed Python code within HTML files** and pass data from Flask to HTML using `render_template()`.

#### 1. Setting Up HTML Templates in the templates/ Folder

In Flask, all HTML files must be placed inside a **templates/** folder.

Steps:

1. Create a Flask project folder.
2. Inside it, create a `templates/` folder.
3. Inside `templates/`, add an HTML file (`index.html`).
4. Use `render_template()` to send the HTML file from Flask.

**Example:****Project Structure:**

```
/flask_project/
|--- app.py
|--- /templates/
|   |--- index.html
```

**app.py**

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def home():
    return render_template('index.html') # Rendering the HTML file

if __name__ == '__main__':
    app.run(debug=True)
```

**templates/index.html**

```
<!DOCTYPE html>
<html>
<head>
    <title>Flask Template</title>
</head>
<body>
    <h1>Welcome to Flask Templates</h1>
```

```
</body>
</html>
```

Now, run app.py and visit <http://127.0.0.1:5000/> to see the page.

## 2. Using Jinja2 {{ variables }}, {% for loops %}, {% if statements %})

Jinja2 allows embedding **Python-like expressions** inside HTML.

### Passing Variables to an HTML Template

Modify app.py:

```
@app.route('/user/<name>')
def user(name):
    return render_template('user.html', username=name)
```

### templates/user.html

```
<!DOCTYPE html>
<html>
<head>
    <title>User Page</title>
</head>
<body>
    <h1>Hello, {{ username }}!</h1> <!-- Using Jinja2 to display data -->
</body>
</html>
```

Visit <http://127.0.0.1:5000/user/John> → Displays Hello, John!

## Using {% for %} Loop

Modify app.py:

```
@app.route('/fruits')
def fruits():
    fruit_list = ['Apple', 'Banana', 'Cherry']
    return render_template('fruits.html', fruits=fruit_list)
```

### templates/fruits.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Fruits List</title>
</head>
<body>
    <h1>Fruits Available:</h1>
    <ul>
        {% for fruit in fruits %}
            <li>{{ fruit }}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

Visit <http://127.0.0.1:5000/fruits> → Displays a list of fruits.

## Using { % if %} Statement

Modify app.py:

```
@app.route('/status/<int:age>')
def status(age):
    return render_template('status.html', age=age)
```

### templates/status.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Status</title>
</head>
<body>
    {% if age >= 18 %}
        <h1>You are an Adult.</h1>
    {% else %}
        <h1>You are a Minor.</h1>
    {% endif %}
</body>
</html>
```

Visit <http://127.0.0.1:5000/status/20> → Displays You are an Adult.

### 3. Creating a Base Template with Template Inheritance

Instead of repeating the same HTML structure for each page, we use **template inheritance**.

#### Steps:

1. Create a **base.html** with common structure.
2. Extend it in other templates using `{% extends 'base.html' %}`.

#### **templates/base.html**

```
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}My Website{% endblock %}</title>
</head>
<body>
    <nav>
        <a href="/">Home</a> |
        <a href="/about">About</a>
    </nav>
    <hr>
    {% block content %}{% endblock %}
</body>
</html>
```

#### **templates/about.html**

```
{% extends 'base.html' %}

{% block title %}About Us{% endblock %}
```

```
{% block content %}
<h1>About Our Website</h1>
<p>This is a sample Flask website.</p>
{% endblock %}
```

Modify app.py:

```
@app.route('/about')
def about():
    return render_template('about.html')
```

Visit <http://127.0.0.1:5000/about> → Renders about.html using base.html.

#### **4. Sending Data from Flask to HTML using render\_template()**

You can pass **multiple values** from Flask to the template.

Modify app.py:

```
@app.route('/profile')
def profile():
    user_info = {"name": "Alice", "age": 25, "city": "New York"}
    return render_template('profile.html', user=user_info)
```

#### **templates/profile.html**

```
<!DOCTYPE html>
<html>
```

```

<head>
    <title>User Profile</title>
</head>
<body>
    <h1>User Profile</h1>
    <p>Name: {{ user.name }}</p>
    <p>Age: {{ user.age }}</p>
    <p>City: {{ user.city }}</p>
</body>
</html>

```

Visit [\*\*http://127.0.0.1:5000/profile\*\*](http://127.0.0.1:5000/profile) → Displays user details dynamically.

## **5. Static Files (CSS, JS, Images) in the static/ Folder**

In Flask, all **CSS, JavaScript, and Images** should be stored inside the **static/ folder**.

### **Steps:**

1. Create a **static/** folder.
2. Place your **CSS, JS, and images** inside it.
3. Link them using **url\_for('static', filename='path')**.

### **Example:**

#### **Project Structure:**

```

/flask_project/
|—— app.py
|—— /static/
|   |—— style.css

```

```
|   └── script.js  
|   └── logo.png  
└── /templates/  
    └── index.html
```

### **static/style.css**

```
body {  
    background-color: lightblue;  
    font-family: Arial, sans-serif;  
}
```

### **templates/index.html**

```
<!DOCTYPE html>  
<html>  
<head>  
    <title>Flask Static Files</title>  
    <link rel="stylesheet" type="text/css" href="{{ url_for('static',  
filename='style.css') }}">  
</head>  
<body>  
    <h1>Welcome to Flask</h1>  
      
</body>  
</html>
```

Now, the **CSS and images** will be loaded correctly in the Flask app.

## Summary

- ✓ **Rendering Templates:** Store HTML files in templates/ and use render\_template().
- ✓ **Jinja2 Templating:** Use {{ variables }}, {% for %}, and {% if %} to insert dynamic content.
- ✓ **Template Inheritance:** Use {% extends 'base.html' %} to avoid code repetition.
- ✓ **Passing Data to Templates:** Use render\_template('file.html', key=value).
- ✓ **Static Files:** Store CSS, JS, and images in static/ and use url\_for() to access them.

## Mini Project 1: Student Management System (Using Flask and Jinja2)

### Project Overview:

Create a simple Student Management System where users can:

- ✓ View a list of students.
- ✓ Add a new student.
- ✓ Use Jinja2 templating to display student data dynamically.
- ✓ Utilize template inheritance for a consistent layout.
- ✓ Use static CSS for styling.

## Step 1: Project Structure

```
/student_management/  
|__ app.py  
|__ /templates/  
| |__ base.html  
| |__ index.html  
| |__ add_student.html  
|__ /static/  
| |__ style.css
```

## Step 2: Install Flask

Run the following command:

```
pip install flask
```

## Step 3: Create app.py (Flask Application Code)

```
from flask import Flask, render_template, request  
  
app = Flask(__name__)  
  
# Sample student data  
students = [  
    {"id": 1, "name": "Alice", "age": 20, "course": "Computer Science"},  
    {"id": 2, "name": "Bob", "age": 22, "course": "Mechanical Engineering"},  
]  
  
@app.route('/')  
def home():
```

```

return render_template('index.html', students=students)
@app.route('/add', methods=['GET', 'POST'])
def add_student():
    if request.method == 'POST':
        name = request.form['name']
        age = request.form['age']
        course = request.form['course']
        students.append({"id": len(students) + 1, "name": name, "age": int(age),
"course": course})
    return render_template('add_student.html')

if __name__ == '__main__':
    app.run(debug=True)

```

#### **Step 4: Create templates/base.html (Template Inheritance)**

```

<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}Student Management{% endblock %}</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
</head>
<body>
    <nav>
        <a href="/">Home</a> | <a href="/add">Add Student</a>
    </nav>
    <hr>
    {% block content %}{% endblock %}
</body>
</html>

```

**Step 5: Create templates/index.html (Show Student List)**

```
{% extends 'base.html' %}

{% block title %}Student List{% endblock %}

{% block content %}
<h1>Student List</h1>
<table border="1">
<tr>
    <th>ID</th>
    <th>Name</th>
    <th>Age</th>
    <th>Course</th>
</tr>
{% for student in students %}
<tr>
    <td>{{ student.id }}</td>
    <td>{{ student.name }}</td>
    <td>{{ student.age }}</td>
    <td>{{ student.course }}</td>
</tr>
{% endfor %}
</table>
{% endblock %}
```

**Step 6: Create templates/add\_student.html (Form to Add Students)**

```
{% extends 'base.html' %}

{% block title %}Add Student{% endblock %}

{% block content %}
<h1>Add a New Student</h1>
<form method="POST">
    <label>Name:</label>
    <input type="text" name="name" required><br><br>

    <label>Age:</label>
    <input type="number" name="age" required><br><br>

    <label>Course:</label>
    <input type="text" name="course" required><br><br>

    <button type="submit">Add Student</button>
</form>
{% endblock %}
```

**Step 7: Create static/style.css (CSS for Styling)**

```
body {
    font-family: Arial, sans-serif;
}

nav {
    background-color: #333;
    padding: 10px;
```

1150

**Private & Confidential : Vetri Technology Solutions**

```
}
```

```
nav a {
```

```
    color: white;
```

```
    margin-right: 15px;
```

```
    text-decoration: none;
```

```
}
```

### **Running the Project**

1. Open terminal, navigate to project folder, and run: python app.py
2. Open <http://127.0.0.1:5000/> in a browser.

## **Mini Project 2: Blog Website with Dynamic Content**

### **Project Overview:**

- ✓ Display a list of blog posts.
- ✓ Show individual blog post details dynamically.
- ✓ Use Jinja2 templating for dynamic content.
- ✓ Use static CSS for styling.

### **Step 1: Project Structure**

```
/flask_blog/  
|—— app.py  
|—— /templates/  
|   |—— base.html
```

```
|   └── index.html  
|   └── post.html  
└── /static/  
    └── style.css
```

## Step 2: Install Flask

Run:

```
pip install flask
```

## Step 3: Create app.py

```
from flask import Flask, render_template  
  
app = Flask(__name__)  
  
# Sample blog data  
posts = [  
    {"id": 1, "title": "Introduction to Flask", "content": "Flask is a lightweight Python  
framework."},  
    {"id": 2, "title": "Understanding Jinja2", "content": "Jinja2 helps in dynamic  
templating in Flask."},  
]  
  
@app.route('/')  
def home():  
    return render_template('index.html', posts=posts)  
  
@app.route('/post/<int:post_id>')  
def post_detail(post_id):
```

```
post = next((post for post in posts if post["id"] == post_id), None)
return render_template('post.html', post=post)

if __name__ == '__main__':
    app.run(debug=True)
```

#### **Step 4: Create templates/base.html**

```
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}My Blog{% endblock %}</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
</head>
<body>
    <nav>
        <a href="/">Home</a>
    </nav>
    <hr>
    {% block content %}{% endblock %}
</body>
</html>
```

#### **Step 5: Create templates/index.html**

```
{% extends 'base.html' %}

{% block title %}Blog Posts{% endblock %}

{% block content %}
    <h1>Recent Blog Posts</h1>
```

```

<ul>
    {% for post in posts %}
        <li>
            <a href="{{ url_for('post_detail', post_id=post.id) }}>{{ post.title }}</a>
        </li>
    {% endfor %}
</ul>
{% endblock %}

```

### **Step 6: Create templates/post.html**

```

{% extends 'base.html' %}

{% block title %}{{ post.title }}{% endblock %}

{% block content %}
    <h1>{{ post.title }}</h1>
    <p>{{ post.content }}</p>
{% endblock %}

```

### **Step 7: Create static/style.css**

```

body {
    font-family: Arial, sans-serif;
}

```

```

nav {
    background-color: #222;
    padding: 10px;
}

```

```
nav a {
```

```
color: white;  
text-decoration: none;  
margin-right: 10px;  
}
```

## Running the Project

1. Open terminal, navigate to project folder, and run: python app.py
2. Open <http://127.0.0.1:5000/> in a browser.

## Mini Project: Task Management System (Using Flask & Jinja2 Templating)

### ◆ Project Overview:

Create a simple **Task Management System** where users can:

- ✓ View a list of tasks.
- ✓ Add new tasks.
- ✓ Mark tasks as completed.
- ✓ Utilize **Jinja2** templating for dynamic content.
- ✓ Use **template inheritance** for a consistent layout.
- ✓ Store **static files (CSS, JS, Images)** for styling.

### 13 Tasks :

- 1. Set up the Flask project structure**
  - a. Create the required folders (templates/, static/).
  - b. Organize files properly.
- 2. Install Flask and create app.py**
  - a. Install Flask using pip install flask.
  - b. Create a basic Flask app (app.py).
- 3. Create a route to display the list of tasks**
  - a. Define a function to fetch and display tasks.
  - b. Use render\_template() to send data from Flask to HTML.
- 4. Set up the templates/base.html for template inheritance**
  - a. Create a **base template** with a navigation menu.
  - b. Use {% block content %} for other pages to extend it.
- 5. Create templates/index.html to list tasks dynamically**
  - a. Display tasks using a {% for task in tasks %} loop.
  - b. Show task details dynamically.
- 6. Add functionality to create a new task**
  - a. Create a form in add\_task.html using **Jinja2**.
  - b. Use **request.form** to get user input.
- 7. Handle form submission using Flask's POST method**
  - a. Capture form data in Flask.
  - b. Add the new task to the task list.
- 8. Use Jinja2 conditions ({{% if %}}) to show completed tasks**
  - a. Display **completed** tasks differently (e.g., strike-through text).
  - b. Use {{% if task.completed %}} ✓ {{% endif %}} for status.
- 9. Implement task completion functionality**
  - a. Create a button to mark a task as completed.
  - b. Update the task list dynamically.
- 10. Use url\_for() to generate dynamic URLs**
  - a. Use url\_for('route\_name') instead of hardcoded links.

- b. Apply url\_for() to navigation and form actions.

### 11. Create and link a static CSS file for styling

- a. Add a **static/style.css** file.
- b. Link it in **base.html** using: <link rel="stylesheet" href="{{ url\_for('static', filename='style.css') }}>

### 12. Use static images and JavaScript in the project

- a. Store images in /static/images/.
- b. Add a logo in base.html.
- c. Use JavaScript for interactivity (e.g., a delete confirmation).

### 13. Test the project and fix any issues

- a. Run the Flask app (python app.py).
- b. Ensure that all functionalities work correctly.

## Mini Project 1: Employee Directory (Using Flask & Jinja2 Templating)

Project Requirements:

- Create an **Employee Directory** where users can:
  - View a list of employees with details (name, department, and position).
  - Search for employees by department using query parameters.
  - Add a new employee using an HTML form and display it dynamically.
  - Use **Jinja2 templating** for looping through employee data ({{ for %}) and conditional rendering ({{ if %}}).
  - Implement a **base template (base.html)** for consistent design across pages.
  - Store **static assets (CSS, images, JS)** in the static/ folder.

## Mini Project 2: Online Product Catalog (Using Flask & Jinja2 Templating)

Project Requirements:

- Develop an Online Product Catalog where users can:
  - View a list of products with their names, prices, and images.
  - Click on a product to view more details on a separate page (product.html).
  - Use Jinja2 templating to display product data dynamically ({{% for product in products %}}).
  - Create a base template (base.html) for styling consistency.
  - Store static files (CSS, JS, images) in the static/ folder.

## Day 78

### Working with Forms & User Input in Flask

Forms are a crucial part of web applications, allowing users to submit data. Flask provides two ways to handle forms:

1. Basic HTML Forms with Flask
2. Flask-WTF (Flask extension for handling forms easily)

We'll use Flask-WTF as it simplifies form handling, validation, and CSRF protection.

## 1. Handling User Input Using Flask Forms (Flask-WTF)

### What is Flask-WTF?

Flask-WTF is a Flask extension that integrates WTForms into Flask, making form handling easier. It provides features like:

- Form Validation
- CSRF Protection
- Error Handling

### Installation

To use Flask-WTF, install it using:

```
pip install flask-wtf
```

### Syntax: Creating a Flask-WTF Form

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, SubmitField
from wtforms.validators import DataRequired, Length, Email

class LoginForm(FlaskForm):
    email = StringField('Email', validators=[DataRequired(), Email()])
    password = PasswordField('Password', validators=[DataRequired(),
Length(min=6)])
    submit = SubmitField('Login')
```

### Real-Life Example: Creating a Login Form

#### Step 1: Setup Flask App

```
from flask import Flask, render_template
```

```
app = Flask(__name__)
app.config['SECRET_KEY'] = 'your_secret_key_here' # Required for CSRF
```

1159

**Private & Confidential : Vetri Technology Solutions**

protection

```
if __name__ == '__main__':
    app.run(debug=True)
```

### **Step 2: Creating the Flask-WTF Form (forms.py)**

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, SubmitField
from wtforms.validators import DataRequired, Email, Length

class LoginForm(FlaskForm):
    email = StringField('Email', validators=[DataRequired(), Email()])
    password = PasswordField('Password', validators=[DataRequired(),
Length(min=6)])
    submit = SubmitField('Login')
```

### **Step 3: Handling User Input & Form Validation (app.py)**

```
from flask import Flask, render_template, request, flash, redirect, url_for
from forms import LoginForm

app = Flask(__name__)
app.config['SECRET_KEY'] = 'your_secret_key_here'

@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        email = form.email.data
        password = form.password.data
        flash(f'Login successful for {email}', 'success')
```

```
return redirect(url_for('home'))  
return render_template('login.html', form=form)  
  
@app.route('/')  
def home():  
    return "Welcome to the Home Page!"  
  
if __name__ == '__main__':  
    app.run(debug=True)
```

#### Step 4: Creating the Login Page (templates/login.html)

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
    <title>Login Page</title>  
</head>  
<body>  
    <h2>Login Form</h2>  
    {% for message in get_flashed_messages() %}  
        <p style="color:green;">{{ message }}</p>  
    {% endfor %}  
  
    <form method="POST">  
        {{ form.hidden_tag() }} <!-- CSRF Token -->  
        <p>{{ form.email.label }} {{ form.email }}</p>  
        <p>{{ form.password.label }} {{ form.password }}</p>  
        <p>{{ form.submit }}</p>  
    </form>  
</body>  
</html>
```

## 2. Validating Form Data (`validators.Length`, `validators.Email`, etc.)

Flask-WTF provides **built-in validators** to check form data.

Validator	Description
<code>DataRequired()</code>	Ensures field is not empty
<code>Email()</code>	Validates email format
<code>Length(min, max)</code>	Ensures length constraints

### Example:

```
from wtforms.validators import DataRequired, Length, Email
```

```
email = StringField('Email', validators=[DataRequired(), Email()])
password = PasswordField('Password', validators=[DataRequired(),
Length(min=6)])
```

## 3. Flash Messages (`flash()`, `get_flashed_messages()`)

**Flash messages** are used to display feedback to users.

### Syntax:

```
flash('Your message here', 'category')
```

- Categories: 'success', 'error', 'warning'

### Example:

```
flash('Login successful!', 'success')
flash('Invalid email or password!', 'error')
```

### Displaying Flash Messages in HTML:

```
{% for message in get_flashed_messages() %}
<p>{{ message }}</p>
{% endfor %}
```

## 4. Handling Form Errors

If the user submits incorrect data, Flask-WTF stores errors.

### Example:

```
{% for field, errors in form.errors.items() %}  
  {% for error in errors %}  
    <p style="color: red;">{{ field }}: {{ error }}</p>  
  {% endfor %}  
{% endfor %}
```

### Summary

- ✓ Flask-WTF makes form handling easy
- ✓ Validators ensure data correctness
- ✓ Flash messages provide user feedback
- ✓ Form errors can be displayed dynamically

Let me know if you need further clarifications!

## Mini Project 1: User Registration System

A web application where users can register with an email, username, and password. The form will include:

- ✓ Validation (Email, Password Length, Required Fields)
- ✓ Flash Messages for Success/Error Feedback
- ✓ Handling Form Errors

### Step 1: Install Required Packages

Ensure you have Flask and Flask-WTF installed:

`pip install flask flask-wtf`

**Step 2: Create Flask App (app.py)**

```
from flask import Flask, render_template, redirect, url_for, flash
from forms import RegistrationForm

app = Flask(__name__)
app.config['SECRET_KEY'] = 'your_secret_key' # Required for CSRF protection

@app.route('/register', methods=['GET', 'POST'])
def register():
    form = RegistrationForm()
    if form.validate_on_submit():
        flash(f'Registration successful for {form.username.data}!', 'success')
        return redirect(url_for('welcome'))
    return render_template('register.html', form=form)

@app.route('/welcome')
def welcome():
    return "<h2>Welcome! Your account has been created successfully.</h2>"

if __name__ == '__main__':
    app.run(debug=True)
```

**Step 3: Create Form Handling (forms.py)**

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, SubmitField
from wtforms.validators import DataRequired, Email, Length

class RegistrationForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired(), Length(min=3, max=15)])
```

```
email = StringField('Email', validators=[DataRequired(), Email()])
password = PasswordField('Password', validators=[DataRequired(),
Length(min=6)])
submit = SubmitField('Register')
```

**Step 4: Create HTML Form (templates/register.html)**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Register</title>
</head>
<body>
    <h2>User Registration</h2>
    {% for message in get_flashed_messages() %}
        <p style="color:green;">{{ message }}</p>
    {% endfor %}
    <form method="POST">
        {{ form.hidden_tag() }}
        <p>{{ form.username.label }} {{ form.username }}</p>
        <p>{{ form.email.label }} {{ form.email }}</p>
        <p>{{ form.password.label }} {{ form.password }}</p>
        <p>{{ form.submit }}</p>
    </form>
    {% for field, errors in form.errors.items() %}
        {% for error in errors %}
            <p style="color: red;">{{ field }}: {{ error }}</p>
        {% endfor %}
    {% endfor %}
</body>
</html>
```

## Step 5: Run the App

```
python app.py
```

Visit <http://127.0.0.1:5000/register> in your browser and test the form!

## Mini Project 2: Contact Form with Message Submission

A contact form where users enter their name, email, and message. After submission, a flash message confirms the message was sent.

### Step 1: Install Required Packages

```
pip install flask flask-wtf
```

### Step 2: Create Flask App (app.py)

```
from flask import Flask, render_template, flash, redirect, url_for
from forms import ContactForm
```

```
app = Flask(__name__)
app.config['SECRET_KEY'] = 'your_secret_key'

@app.route('/contact', methods=['GET', 'POST'])
def contact():
    form = ContactForm()
    if form.validate_on_submit():
        flash(f'Thank you, {form.name.data}! Your message has been received.')
        'success')
        return redirect(url_for('home'))
    return render_template('contact.html', form=form)
```

```
@app.route('/')
def home():
    return "<h2>Welcome to the Contact Page!</h2>

if __name__ == '__main__':
    app.run(debug=True)
```

**Step 3: Create the Contact Form (forms.py)**

```
from flask_wtf import FlaskForm
from wtforms import StringField, TextAreaField, SubmitField
from wtforms.validators import DataRequired, Email, Length

class ContactForm(FlaskForm):
    name = StringField('Name', validators=[DataRequired(), Length(min=2, max=50)])
    email = StringField('Email', validators=[DataRequired(), Email()])
    message = TextAreaField('Message', validators=[DataRequired(), Length(min=10, max=500)])
    submit = SubmitField('Send Message')
```

**Step 4: Create the HTML Form (templates/contact.html)**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Contact Us</title>
</head>
<body>
    <h2>Contact Form</h2>
    {% for message in get_flashed_messages() %}
        <p style="color:green;">{{ message }}</p>
```

```
{% endfor %}

<form method="POST">
    {{ form.hidden_tag() }}
    <p>{{ form.name.label }} {{ form.name }}</p>
    <p>{{ form.email.label }} {{ form.email }}</p>
    <p>{{ form.message.label }} {{ form.message }}</p>
    <p>{{ form.submit }}</p>
</form>

{% for field, errors in form.errors.items() %}
    {% for error in errors %}
        <p style="color: red;">{{ field }}: {{ error }}</p>
    {% endfor %}
    {% endfor %}
</body>
</html>
```

### Step 5: Run the App

python app.py

Visit <http://127.0.0.1:5000/contact> and try sending a message.

### Summary

- ✓ Mini Project 1: User Registration System
- ✓ Mini Project 2: Contact Form with Message Submission
- ✓ Uses Flask-WTF, Form Validation, Flash Messages, and Error Handling

Let me know if you need modifications!

## Real-Life Mini Project: Job Application Form

A Job Application Portal where users can apply for a job by entering their name, email, phone number, resume (file upload), and cover letter. The application form will include:

- ✓ Validation (Email, Phone Number Format, Required Fields, File Upload)
- ✓ Flash Messages for Success/Error Feedback
- ✓ Handling Form Errors

## Day 78 Tasks

1. Install Required Dependencies
  - a. Install Flask and Flask-WTF to handle forms.
2. Create a Flask App (app.py)
  - a. Set up a basic Flask application with routing.
3. Configure Flask Secret Key for CSRF Protection
  - a. Use app.config['SECRET\_KEY'] to enable security features.
4. Create a Form Class (forms.py)
  - a. Define a JobApplicationForm class with fields: name, email, phone, resume, and cover\_letter.
5. Add Validation Rules to Form Fields
  - a. Ensure name is required, email is in a valid format, phone follows a pattern, and resume only allows .pdf and .docx.
6. Create the Job Application Route (/apply)
  - a. Set up a POST request to handle form submissions.
7. Render the Job Application Form (templates/apply.html)
  - a. Display input fields and submit button using render\_template().
8. Handle Form Submissions in the Flask Route
  - a. If the form is valid, flash a success message and redirect.

9. Handle File Uploads (Resume Upload)
  - a. Store the uploaded file in a designated uploads folder.
10. Implement Flash Messages for Success & Error Handling
  - a. Use flash() for feedback messages after submission.
11. Validate and Display Error Messages for Invalid Inputs
  - a. Show error messages if the email is incorrect or the phone number format is wrong.
12. Create a Confirmation Page (/success)
  - a. Display a success message after a valid submission.
13. Run the Flask App & Test the Job Application Form
  - a. Start the Flask development server and ensure the form works as expected.

## Real-Life Mini Project Requirements (Without Solutions)

### 1. Online Event Registration Form

Create an **Event Registration System** where users can register for an event by entering:

- ✓ Full Name (Required, min length 3 characters)
- ✓ Email (Must be in valid format)
- ✓ Phone Number (Numeric, 10 digits)
- ✓ Event Selection (Dropdown to choose an event)
- ✓ Comments (Optional)

#### Validation Rules:

- Name must be at least 3 characters long.
- Email must be in valid format.
- Phone number must be exactly 10 digits.

#### Features:

- If registration is successful, show a flash success message.

- If validation fails, display form error messages.
- Redirect users to a confirmation page after a successful registration.

## 2. Customer Support Ticket System

Develop a Customer Support System where users can submit support tickets for assistance. The form should include:

- ✓ User Name (Required)
- ✓ Email Address (Valid format required)
- ✓ Issue Type (Dropdown with options like "Technical Issue," "Billing Issue," "General Inquiry")
- ✓ Description of the Issue (Minimum 20 characters)

### Validation Rules:

- Name and issue description fields are required.
- Email must be valid.
- Issue description must be at least 20 characters long.

### Features:

- Show flash messages for success/error feedback.
- Display validation errors if inputs are incorrect.
- Redirect to a ticket submission success page.

## Day 79

### Connecting Flask with Databases (MySQL)

In this guide, we will learn how to connect Flask with MySQL using Flask-SQLAlchemy, an ORM (Object Relational Mapping) tool for working with databases in Flask applications. We will cover:

- ✓ Setting up MySQL with Flask

- ✓ Using Flask-SQLAlchemy for ORM
- ✓ Creating Database Models (db.Model)
- ✓ Performing CRUD Operations (Create, Read, Update, Delete)

## 1. Setting up MySQL with Flask

### What is MySQL? Why Use It with Flask?

MySQL is a relational database management system (RDBMS) that stores data in structured tables. It is widely used in web applications due to its speed, reliability, and scalability.

Flask does not have built-in database support, so we use Flask-SQLAlchemy, which allows us to interact with MySQL using Python instead of raw SQL queries.

### Install Required Packages

Before connecting Flask with MySQL, install the necessary libraries:

```
pip install flask flask-sqlalchemy flask-mysql flask-migrate pymysql
```

- flask → Flask framework
- flask-sqlalchemy → ORM for Flask
- flask-mysql → MySQL connector
- flask-migrate → Database migrations
- pymysql → MySQL client for Python

## 2. Using Flask-SQLAlchemy for ORM

### What is ORM?

Object Relational Mapping (ORM) converts database tables into Python objects. Instead of writing raw SQL queries, we define Python classes (models) that represent database tables.

## 3. Creating Database Models (db.Model)

### Database Configuration in Flask

We need to tell Flask how to connect to the MySQL database.

#### config.py (Database Configuration)

```
import os

class Config:
    SQLALCHEMY_DATABASE_URI =
        'mysql+pymysql://username:password@localhost/db_name'
    SQLALCHEMY_TRACK_MODIFICATIONS = False
```

Replace username, password, and db\_name with your actual MySQL credentials.

### Initialize Flask and Database

#### app.py (Main Flask App)

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config.from_object('config.Config')
# Initialize SQLAlchemy
db = SQLAlchemy(app)
```

- ◆ `SQLALCHEMY_DATABASE_URI` → Defines the MySQL connection string.
- ◆ `SQLAlchemy(app)` → Connects Flask to MySQL.

## Define a Database Model

Now, let's create a User model to store user details.

### **models.py (Define Models)**

```
from app import db
```

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100), nullable=False)
    email = db.Column(db.String(100), unique=True, nullable=False)
    age = db.Column(db.Integer, nullable=False)
```

```
def __repr__(self):
    return f"User('{self.name}', '{self.email}', '{self.age}')"
```

- ◆ `db.Model` → Base class for models.
- ◆ `id` → Primary key (auto-incrementing).
- ◆ `name` → String (100 characters).
- ◆ `email` → Unique field (no duplicate emails).
- ◆ `age` → Integer field.

## Create Database and Tables

Run these commands in Python shell (or add a script to initialize the database).

```
from app import db
db.create_all()
```

This will create a users table in MySQL.

## 4. Performing CRUD Operations (Create, Read, Update, Delete)

Let's create routes to perform CRUD operations in Flask.

### 1. Insert Data (CREATE)

#### **routes.py (Insert Data into MySQL)**

```
from flask import Flask, request, jsonify
from app import db, app
from models import User
```

```
@app.route('/add_user', methods=['POST'])
def add_user():
    data = request.json # Get JSON data from request
    new_user = User(name=data['name'], email=data['email'], age=data['age'])

    db.session.add(new_user) # Add to database
    db.session.commit() # Save changes

    return jsonify({"message": "User added successfully!"})
```

✓ `request.json` → Fetch JSON data from the request.

✓ `db.session.add()` → Adds new user to MySQL.

✓ `db.session.commit()` → Saves changes.

### 2. Retrieve Data (READ)

#### **routes.py (Fetch Users from MySQL)**

```
@app.route('/get_users', methods=['GET'])
def get_users():
    users = User.query.all() # Fetch all users
```

```

output = []

for user in users:
    user_data = {'id': user.id, 'name': user.name, 'email': user.email, 'age':
user.age}
    output.append(user_data)

return jsonify(output)

```

- ✓ User.query.all() → Retrieves all users from the database.

### **3. Update Data (UPDATE)**

#### **routes.py (Update User Data in MySQL)**

```

@app.route('/update_user/<int:id>', methods=['PUT'])
def update_user(id):
    user = User.query.get(id) # Find user by ID

    if not user:
        return jsonify({"message": "User not found!"})

    data = request.json
    user.name = data.get('name', user.name)
    user.email = data.get('email', user.email)
    user.age = data.get('age', user.age)

    db.session.commit() # Save changes
    return jsonify({"message": "User updated successfully!"})

```

- ✓ User.query.get(id) → Fetch user by ID.

- ✓ db.session.commit() → Saves updated user details.

#### 4. Delete Data (DELETE)

##### routes.py (Delete User from MySQL)

```
@app.route('/delete_user/<int:id>', methods=['DELETE'])
def delete_user(id):
    user = User.query.get(id) # Find user by ID

    if not user:
        return jsonify({"message": "User not found!"})

    db.session.delete(user) # Delete user from database
    db.session.commit() # Save changes

    return jsonify({"message": "User deleted successfully!"})
```

✓ db.session.delete(user) → Deletes user from MySQL.

#### Final Steps: Running the Flask App

Run the Flask app to test the API.

```
python app.py
```

The API will be available at:

- POST /add\_user → Add user
- GET /get\_users → Get all users
- PUT /update\_user/<id> → Update user
- DELETE /delete\_user/<id> → Delete user

## Conclusion

- ✓ We successfully connected Flask with MySQL using Flask-SQLAlchemy.
- ✓ We created a User model and performed CRUD operations.
- ✓ The app now allows adding, reading, updating, and deleting users in MySQL.

## Mini Project 1: Employee Management System

### Project Overview

A simple Employee Management System using Flask and MySQL to:

- Add new employees
- View all employees
- Update employee details
- Delete employees

### Step 1: Install Dependencies

```
pip install flask flask-sqlalchemy flask-mysql pymysql
```

### Step 2: Configure Flask & MySQL

#### config.py

```
class Config:  
    SQLALCHEMY_DATABASE_URI =  
        'mysql+pymysql://root:password@localhost/employee_db'  
    SQLALCHEMY_TRACK_MODIFICATIONS = False
```

**Step 3: Initialize Flask & SQLAlchemy****app.py**

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config.from_object('config.Config')

db = SQLAlchemy(app)
```

**Step 4: Create Database Model****models.py**

```
from app import db

class Employee(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100), nullable=False)
    email = db.Column(db.String(100), unique=True, nullable=False)
    department = db.Column(db.String(100), nullable=False)
```

Run:

```
from app import db
db.create_all()
```

**Step 5: Implement CRUD Operations****Create Employee (POST)**

```
@app.route('/add_employee', methods=['POST'])
def add_employee():
    data = request.json
```

```

new_employee = Employee(name=data['name'], email=data['email'],
department=data['department'])
db.session.add(new_employee)
db.session.commit()
return jsonify({"message": "Employee added!"})

```

### **Read Employees (GET)**

```

@app.route('/get_employees', methods=['GET'])
def get_employees():
    employees = Employee.query.all()
    return jsonify([{"id": e.id, "name": e.name, "email": e.email, "department": e.department} for e in employees])

```

### **Update Employee (PUT)**

```

@app.route('/update_employee/<int:id>', methods=['PUT'])
def update_employee(id):
    employee = Employee.query.get(id)
    if not employee:
        return jsonify({"message": "Employee not found!"})

    data = request.json
    employee.name = data.get('name', employee.name)
    employee.email = data.get('email', employee.email)
    employee.department = data.get('department', employee.department)

    db.session.commit()
    return jsonify({"message": "Employee updated!"})

```

### **Delete Employee (DELETE)**

```

@app.route('/delete_employee/<int:id>', methods=['DELETE'])
def delete_employee(id):
    employee = Employee.query.get(id)

```

```
if not employee:  
    return jsonify({"message": "Employee not found!"})  
  
db.session.delete(employee)  
db.session.commit()  
return jsonify({"message": "Employee deleted!"})
```

### Step 6: Run Flask App

python app.py

Test APIs:

- POST /add\_employee → Add employee
- GET /get\_employees → List employees
- PUT /update\_employee/<id> → Update employee
- DELETE /delete\_employee/<id> → Delete employee

✓ Employee Management System is now ready!

## Mini Project 2: Student Course Registration System

### Project Overview

A Student Course Registration System using Flask and MySQL.

- Add students
- Register students for courses
- View enrolled courses
- Update student details
- Remove students

### Step 1: Install Dependencies

```
pip install flask flask-sqlalchemy flask-mysql pymysql
```

### Step 2: Configure Flask & MySQL

#### config.py

```
class Config:  
    SQLALCHEMY_DATABASE_URI =  
        'mysql+pymysql://root:password@localhost/course_db'  
    SQLALCHEMY_TRACK_MODIFICATIONS = False
```

### Step 3: Initialize Flask & SQLAlchemy

#### app.py

```
from flask import Flask  
from flask_sqlalchemy import SQLAlchemy  
  
app = Flask(__name__)  
app.config.from_object('config.Config')  
  
db = SQLAlchemy(app)
```

### Step 4: Create Database Models

#### models.py

```
from app import db  
  
class Student(db.Model):  
    id = db.Column(db.Integer, primary_key=True)  
    name = db.Column(db.String(100), nullable=False)  
    email = db.Column(db.String(100), unique=True, nullable=False)
```

```

class Course(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(100), nullable=False)

class Enrollment(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    student_id = db.Column(db.Integer, db.ForeignKey('student.id'), nullable=False)
    course_id = db.Column(db.Integer, db.ForeignKey('course.id'), nullable=False)

```

Run:

```

from app import db
db.create_all()

```

## **Step 5: Implement CRUD Operations**

### **Add Student (POST)**

```

@app.route('/add_student', methods=['POST'])
def add_student():
    data = request.json
    new_student = Student(name=data['name'], email=data['email'])
    db.session.add(new_student)
    db.session.commit()
    return jsonify({"message": "Student added!"})

```

### **Register Student for Course (POST)**

```

@app.route('/enroll_student', methods=['POST'])
def enroll_student():
    data = request.json
    new_enrollment = Enrollment(student_id=data['student_id'],
                                course_id=data['course_id'])

```

```
db.session.add(new_enrollment)
db.session.commit()
return jsonify({"message": "Student enrolled!"})
```

**Get Students (GET)**

```
@app.route('/get_students', methods=['GET'])
def get_students():
    students = Student.query.all()
    return jsonify([{"id": s.id, "name": s.name, "email": s.email} for s in students])
```

**Get Courses (GET)**

```
@app.route('/get_courses', methods=['GET'])
def get_courses():
    courses = Course.query.all()
    return jsonify([{"id": c.id, "title": c.title} for c in courses])
```

**Get Student Enrollments (GET)**

```
@app.route('/get_enrollments', methods=['GET'])
def get_enrollments():
    enrollments = Enrollment.query.all()
    return jsonify([{"student_id": e.student_id, "course_id": e.course_id} for e in
enrollments])
```

**Update Student (PUT)**

```
@app.route('/update_student/<int:id>', methods=['PUT'])
def update_student(id):
    student = Student.query.get(id)
    if not student:
        return jsonify({"message": "Student not found!"})
```

```

data = request.json
student.name = data.get('name', student.name)
student.email = data.get('email', student.email)

db.session.commit()
return jsonify({"message": "Student updated!"})

```

### **Delete Student (DELETE)**

```

@app.route('/delete_student/<int:id>', methods=['DELETE'])
def delete_student(id):
    student = Student.query.get(id)
    if not student:
        return jsonify({"message": "Student not found!"})

    db.session.delete(student)
    db.session.commit()
    return jsonify({"message": "Student deleted!"})

```

### **Step 6: Run Flask App**

python app.py

Test APIs:

- POST /add\_student → Add student
- POST /enroll\_student → Enroll student in course
- GET /get\_students → View students
- PUT /update\_student/<id> → Update student
- DELETE /delete\_student/<id> → Delete student

✓ Student Course Registration System is now ready!

## Real-Life Mini Project: Online Bookstore Management System

### Project Overview

This Online Bookstore Management System allows users to:

- Add books to the database
- View available books
- Update book details
- Delete books
- Manage customers
- Handle book purchases

### Day 79 Tasks

#### Task 1: Install Flask and Required Packages

- Install Flask and necessary dependencies (flask, flask-sqlalchemy, pymysql).

#### Task 2: Configure Flask and MySQL

- Set up Flask app and configure MySQL connection.
- Use SQLAlchemy to interact with MySQL.

#### Task 3: Create Database Models

- Define Book, Customer, and Order models using Flask-SQLAlchemy.
- Establish relationships between tables (One-to-Many).

#### Task 4: Initialize the Database

- Create database schema and tables by running migrations.

**Task 5: Implement Create Operation (Add Books & Customers)**

- Create API routes or views to add books and customers to the database.
- Validate data before insertion.

**Task 6: Implement Read Operation (Retrieve Books & Customers)**

- Create API endpoints to fetch all books and specific book details.
- Display books in an HTML page using Jinja2 templates.

**Task 7: Implement Update Operation (Modify Book Details)**

- Allow users to update book price, stock, or description.
- Implement update functionality via Flask routes.

**Task 8: Implement Delete Operation (Remove Books)**

- Provide an option to delete books from the database.
- Ensure cascading deletion if required.

**Task 9: Implement Book Purchase Feature**

- Allow customers to place an order.
- Deduct stock quantity upon successful purchase.

**Task 10: Display Orders & Customers**

- List customer orders with book details.
- Fetch and display order history.

**Task 11: Handle Form Validations**

- Validate form inputs using Flask-WTF and SQLAlchemy constraints.
- Prevent duplicate customer emails.

### Task 12: Implement Flash Messages

- Display success/error messages for operations (book added, order placed, etc.).

### Task 13: Deploy the Application

- Run Flask app and test API routes using Postman or Browser.
- Deploy using Gunicorn + Hostinger or Docker.

## **Mini Project 1: Online Appointment Booking System**

Objective: Develop a Flask-based Online Appointment Booking System where users can book appointments with professionals (e.g., doctors, tutors, consultants) and manage their schedules using MySQL.

### **Features:**

- ◆ User Registration & Login: Users can create an account and log in.
- ◆ Add Services: Admin can add different services (e.g., doctor consultation, tutoring).
- ◆ Book Appointment: Users can select a date, time, and service provider to book an appointment.
- ◆ Manage Appointments: Users can view, reschedule, or cancel their appointments.
- ◆ Admin Dashboard: Admin can manage bookings, users, and service providers.
- ◆ Email Notifications: Send confirmation emails for booked or canceled appointments.
- ◆ Search & Filters: Users can search services by category and filter available slots.

## Mini Project 2: Expense Tracker Web App

Objective: Create a Flask-based Expense Tracker that allows users to track their income and expenses using MySQL.

### Features:

- ◆ User Authentication: Users can sign up and log in.
- ◆ Add Expenses: Users can log their expenses with details (amount, category, date, notes).
- ◆ Add Income: Users can track their income sources.
- ◆ View Transaction History: Display all transactions with filtering options (date, category).
- ◆ Edit & Delete Transactions: Users can update or remove entries.
- ◆ Dashboard with Charts: Show spending patterns using visual graphs (using Chart.js or similar).
- ◆ Set Monthly Budget: Users can set a budget limit and get alerts when approaching the limit.

## Day 80

### Flask Authentication & User Management

Flask provides powerful tools for handling user authentication and session management. This allows you to create a secure user registration and login system with features like password hashing, session handling, and authentication checks.

## 1. Implementing User Registration & Login System

### Definition:

A User Registration & Login System allows users to create accounts, log in securely, and access protected resources in a Flask web application.

### Steps to Implement:

1. Set up a Flask project.
2. Configure Flask-SQLAlchemy for database storage.
3. Use Flask-WTF for user input validation.
4. Store hashed passwords using werkzeug.security.
5. Implement Flask-Login for authentication.

## 2. Hashing Passwords using werkzeug.security

### Definition:

Password hashing ensures that passwords are not stored in plain text. Instead, they are converted into a secure hash using werkzeug.security.

### Syntax:

```
from werkzeug.security import generate_password_hash, check_password_hash

hashed_password = generate_password_hash("mypassword",
method="pbkdf2:sha256")
print(hashed_password)
# Checking password
check = check_password_hash(hashed_password, "mypassword")
print(check) # Output: True (if password matches)
```

### Why Use Hashing?

- Prevents storing plain-text passwords.
- Protects against brute-force attacks.
- Even if the database is compromised, passwords remain unreadable.

### **3. Using Flask-Login for Authentication**

#### **Definition:**

Flask-Login helps manage user sessions, keeping users logged in across different pages.

#### **Installation:**

```
pip install flask-login
```

#### **Steps to Implement:**

##### **1. Define a User Model:**

```
from flask_login import UserMixin
```

```
class User(db.Model, UserMixin):
```

```
    id = db.Column(db.Integer, primary_key=True)
```

```
    username = db.Column(db.String(100), unique=True, nullable=False)
```

```
    password = db.Column(db.String(200), nullable=False)
```

##### **2. Initialize Flask-Login:**

```
from flask_login import LoginManager
```

```
login_manager = LoginManager()
```

```
login_manager.init_app(app)
```

```
login_manager.login_view = "login"
```

##### **3. Load Users Dynamically:**

```
@login_manager.user_loader
```

```
def load_user(user_id):
```

```
    return User.query.get(int(user_id))
```

## 4. Session Management (session, login\_required)

### Definition:

- session: Stores user session data (e.g., user ID, preferences).
- login\_required: Restricts access to certain pages unless logged in.

### Example Implementation:

```
from flask import session
```

```
# Storing user data
session["username"] = "john_doe"
```

```
# Accessing session data
username = session.get("username")
```

```
# Removing session data
session.pop("username", None)
```

## 5. Logout Functionality

### Definition:

Logout functionality removes the user session, ensuring security after logout.

### Implementation:

```
from flask_login import logout_user
```

```
@app.route("/logout")
@login_required
def logout():
    logout_user()
    return redirect(url_for("login"))
```

## Step-by-Step Implementation of a Full Flask Authentication System

### 1. Install Required Libraries

```
pip install flask flask-sqlalchemy flask-login flask-wtf werkzeug
```

### 2. Create a Flask App (app.py)

```
from flask import Flask, render_template, redirect, url_for, request, flash
from flask_sqlalchemy import SQLAlchemy
from flask_login import LoginManager, UserMixin, login_user, logout_user,
login_required, current_user
from werkzeug.security import generate_password_hash, check_password_hash

app = Flask(__name__)
app.config["SECRET_KEY"] = "mysecret"
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///users.db"

db = SQLAlchemy(app)
login_manager = LoginManager()
login_manager.init_app(app)
login_manager.login_view = "login"

class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(100), unique=True, nullable=False)
    password = db.Column(db.String(200), nullable=False)

@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))

@app.route("/register", methods=["GET", "POST"])

```

```

def register():
    if request.method == "POST":
        username = request.form["username"]
        password = request.form["password"]
        hashed_password = generate_password_hash(password,
method="pbkdf2:sha256")

        new_user = User(username=username, password=hashed_password)
        db.session.add(new_user)
        db.session.commit()
        flash("Registration successful! Please login.", "success")
        return redirect(url_for("login"))

    return render_template("register.html")

@app.route("/login", methods=["GET", "POST"])
def login():
    if request.method == "POST":
        username = request.form["username"]
        password = request.form["password"]
        user = User.query.filter_by(username=username).first()

        if user and check_password_hash(user.password, password):
            login_user(user)
            return redirect(url_for("dashboard"))

        else:
            flash("Invalid username or password.", "danger")

    return render_template("login.html")

@app.route("/dashboard")

```

```
@login_required
def dashboard():
    return f"Welcome, {current_user.username}!"

@app.route("/logout")
@login_required
def logout():
    logout_user()
    flash("You have been logged out.", "info")
    return redirect(url_for("login"))

if __name__ == "__main__":
    db.create_all()
    app.run(debug=True)
```

## 6. HTML Files (templates/)

### register.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Register</title>
</head>
<body>
    <h2>Register</h2>
    <form method="POST">
        <input type="text" name="username" placeholder="Username" required>
        <input type="password" name="password" placeholder="Password" required>
        <button type="submit">Register</button>
    </form>
```

```
</body>
</html>
```

### **login.html**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Login</title>
</head>
<body>
    <h2>Login</h2>
    <form method="POST">
        <input type="text" name="username" placeholder="Username" required>
        <input type="password" name="password" placeholder="Password"
required>
        <button type="submit">Login</button>
    </form>
</body>
</html>
```

## **7. Running the Application**

### **1. Initialize the Database**

```
python
>>> from app import db
>>> db.create_all()
>>> exit()
```

## 2. Run the Flask App

`python app.py`

- Open <http://127.0.0.1:5000/register> → Register a new user.
- Open <http://127.0.0.1:5000/login> → Log in with your credentials.
- Open <http://127.0.0.1:5000/dashboard> → View the protected dashboard.
- Open <http://127.0.0.1:5000/logout> → Log out.

## Summary

Feature	Explanation
User Registration	Stores hashed passwords in MySQL.
Login System	Uses Flask-Login to manage authentication.
Password Hashing	Ensures security using werkzeug.security.
Session Management	Uses session and login_required to protect pages.
Logout Functionality	Logs out users and redirects to login.

## Mini Project 1: Employee Management System (EMS)

### Project Overview:

A Flask-based Employee Management System where employees can register, log in, and access their dashboard. Admins can manage employee accounts.

### Step-by-Step Implementation

#### Step 1: Set Up Flask Project

1. Create a project folder: `mkdir flask_ems && cd flask_ems`
2. Install dependencies: `pip install flask flask-sqlalchemy flask-login flask-wtf werkzeug`
3. Create `app.py`.

## Step 2: Configure Flask & Database

- Define Flask settings and database:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_login import LoginManager

app = Flask(__name__)
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///employees.db"
app.config["SECRET_KEY"] = "your_secret_key"

db = SQLAlchemy(app)
login_manager = LoginManager()
login_manager.init_app(app)
login_manager.login_view = "login"
```

## Step 3: Create Employee Model

- Define Employee table:

```
from flask_login import UserMixin

class Employee(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100), nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)
    password = db.Column(db.String(200), nullable=False)
```

## Step 4: Set Up Registration & Login Forms

- Create forms.py:

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, SubmitField
```

```

from wtforms.validators import DataRequired, Email, Length

class RegisterForm(FlaskForm):
    name = StringField("Name", validators=[DataRequired()])
    email = StringField("Email", validators=[DataRequired(), Email()])
    password = PasswordField("Password", validators=[DataRequired(),
Length(min=6)])
    submit = SubmitField("Register")

class LoginForm(FlaskForm):
    email = StringField("Email", validators=[DataRequired(), Email()])
    password = PasswordField("Password", validators=[DataRequired()])
    submit = SubmitField("Login")

```

### **Step 5: Implement Registration Route**

```

from flask import render_template, request, redirect, url_for, flash
from werkzeug.security import generate_password_hash
from forms import RegisterForm

@app.route("/register", methods=["GET", "POST"])
def register():
    form = RegisterForm()
    if form.validate_on_submit():
        hashed_password = generate_password_hash(form.password.data,
method="pbkdf2:sha256")
        new_employee = Employee(name=form.name.data, email=form.email.data,
password=hashed_password)
        db.session.add(new_employee)
        db.session.commit()
        flash("Account created successfully! Please log in.", "success")

```

```
    return redirect(url_for("login"))
    return render_template("register.html", form=form)
```

### Step 6: Implement Login Route

```
from flask_login import login_user
from werkzeug.security import check_password_hash
from forms import LoginForm

@app.route("/login", methods=["GET", "POST"])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        employee = Employee.query.filter_by(email=form.email.data).first()
        if employee and check_password_hash(employee.password,
form.password.data):
            login_user(employee)
            return redirect(url_for("dashboard"))
        else:
            flash("Invalid email or password.", "danger")
    return render_template("login.html", form=form)
```

### Step 7: Implement Employee Dashboard

```
from flask_login import login_required, current_user

@app.route("/dashboard")
@login_required
def dashboard():
    return f"Welcome, {current_user.name}!"
```

### **Step 8: Implement Logout**

```
from flask_login import logout_user

@app.route("/logout")
@login_required
def logout():
    logout_user()
    flash("You have been logged out.", "info")
    return redirect(url_for("login"))
```

### **Step 9: Run the Application**

1. Initialize database: python

```
>>> from app import db
>>> db.create_all()
>>> exit()
```

2. Run the Flask app: python app.py

## **Mini Project 2: Student Portal Authentication System**

### **Project Overview:**

A Student Portal where students can register, log in, and view their profile. The system uses Flask-Login for authentication.

### **Step-by-Step Implementation**

#### **Step 1: Set Up Flask Project**

1. Create a new project folder: mkdir student\_portal && cd student\_portal
2. Install dependencies: pip install flask flask-sqlalchemy flask-login flask-wtf werkzeug
3. Create app.py.

**Step 2: Configure Flask & Database**

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_login import LoginManager

app = Flask(__name__)
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///students.db"
app.config["SECRET_KEY"] = "super_secret_key"

db = SQLAlchemy(app)
login_manager = LoginManager()
login_manager.init_app(app)
login_manager.login_view = "login"
```

**Step 3: Create Student Model**

```
from flask_login import UserMixin

class Student(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(100), unique=True, nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)
    password = db.Column(db.String(200), nullable=False)
```

**Step 4: Implement Registration Form**

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, SubmitField
from wtforms.validators import DataRequired, Email, Length
```

```
class RegisterForm(FlaskForm):
    username = StringField("Username", validators=[DataRequired()])
```

```

email = StringField("Email", validators=[DataRequired(), Email()])
password = PasswordField("Password", validators=[DataRequired(),
Length(min=6)])
submit = SubmitField("Register")

```

### **Step 5: Implement Login Form**

```

class LoginForm(FlaskForm):
    email = StringField("Email", validators=[DataRequired(), Email()])
    password = PasswordField("Password", validators=[DataRequired()])
    submit = SubmitField("Login")

```

### **Step 6: Implement Registration Route**

```

from werkzeug.security import generate_password_hash

@app.route("/register", methods=["GET", "POST"])
def register():
    form = RegisterForm()
    if form.validate_on_submit():
        hashed_password = generate_password_hash(form.password.data,
method="pbkdf2:sha256")
        new_student = Student(username=form.username.data,
email=form.email.data, password=hashed_password)
        db.session.add(new_student)
        db.session.commit()
        flash("Registration successful!", "success")
        return redirect(url_for("login"))
    return render_template("register.html", form=form)

```

**Step 7: Implement Login Route**

```
from flask_login import login_user
from werkzeug.security import check_password_hash
@app.route("/login", methods=["GET", "POST"])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        student = Student.query.filter_by(email=form.email.data).first()
        if student and check_password_hash(student.password,
form.password.data):
            login_user(student)
            return redirect(url_for("profile"))
    else:
        flash("Invalid email or password.", "danger")
    return render_template("login.html", form=form)
```

**Step 8: Implement Profile Page**

```
@app.route("/profile")
@login_required
def profile():
    return f"Welcome, {current_user.username}!"
```

**Step 9: Implement Logout**

```
@app.route("/logout")
@login_required
def logout():
    logout_user()
    flash("Logged out successfully.", "info")
    return redirect(url_for("login"))
```

## Step 10: Run the Application

```
python app.py
```

## Summary

- ✓ Project 1: Employee Management System – Employee registration, login, and dashboard.
- ✓ Project 2: Student Portal Authentication – Student authentication with profile access.

## Mini Project: Job Portal Authentication System

### Project Overview:

A Flask-based Job Portal where job seekers can register, log in, and access their profile, while recruiters can post jobs. Authentication and session management will be implemented using Flask-Login.

### Day 80 Tasks

#### Task 1: Set Up Flask Project

- Install Flask and required dependencies.
- Create the necessary folders: templates/, static/, and models/.

#### Task 2: Configure Flask & Database

- Initialize Flask-SQLAlchemy.
- Configure SQLite or MySQL database connection.

### Task 3: Create User Model (Job Seekers & Recruiters)

- Define a User model with fields: id, name, email, password, role (job\_seeker/recruiter).
- Implement UserMixin for Flask-Login.

### Task 4: Set Up User Registration Form

- Use Flask-WTF to create a registration form.
- Validate user input (name, email, password length).

### Task 5: Implement Password Hashing

- Use werkzeug.security.generate\_password\_hash() to encrypt passwords before storing them.

### Task 6: Implement User Registration Route

- Save user details in the database after form validation.
- Redirect users to the login page after successful registration.

### Task 7: Set Up Login Form & Authentication

- Create a login form using Flask-WTF.
- Validate credentials using check\_password\_hash().

### Task 8: Implement Login Route & Session Management

- Authenticate users using Flask-Login.
- Create a session and redirect them to their respective dashboard (job\_seeker\_dashboard or recruiter\_dashboard).

#### Task 9: Protect Routes with login\_required

- Restrict access to dashboards unless the user is logged in.

#### Task 10: Implement Logout Functionality

- Use logout\_user() to clear the session.
- Redirect users to the login page after logout.

#### Task 11: Flash Messages for User Feedback

- Show success messages for registration, login, and logout.
- Display error messages for incorrect credentials or form errors.

#### Task 12: Implement User Profile Section

- Allow users to update their profile details.
- Ensure only logged-in users can access the profile page.

#### Task 13: Deploy the Job Portal

- Run the Flask app locally and test authentication.
- Deploy on Hostinger, Heroku, or PythonAnywhere.

### **Real-Life Mini Project 1: Online Course Enrollment System**

#### **Project Requirement:**

Develop an Online Course Enrollment System where students and instructors can register, log in, and manage their profiles.

- **User Roles:**
  - **Students:** Browse available courses, enroll, and track progress.
  - **Instructors:** Create and manage courses, view enrolled students.

- **Authentication Features:**

- Secure User Registration and Login System using Flask-WTF.
- Password Hashing using werkzeug.security.
- Flask-Login for authentication and session management.
- Role-based access: Students can only access their enrolled courses, and instructors can only edit their courses.
- Logout Functionality to end user sessions.

## **Real-Life Mini Project 2: Doctor Appointment Booking System**

### **Project Requirement:**

Build a Doctor Appointment Booking System where patients can book appointments, and doctors can manage their schedules.

- **User Roles:**

- Patients: Register, log in, book appointments, and view appointment history.
- Doctors: Register, log in, manage appointments, and update availability.

- **Authentication Features:**

- Implement Flask-WTF forms for secure user registration.
- Encrypt passwords using werkzeug.security.
- Use Flask-Login to manage user sessions.
- Restrict access to doctor and patient dashboards using login\_required().
- Logout Functionality to securely end sessions.

# Day 81

## Flask API Development (REST API with Flask & Flask-RESTful)

### Introduction to REST APIs in Flask

#### What is a REST API?

A **REST API (Representational State Transfer Application Programming Interface)** allows communication between different applications over the internet using HTTP methods like **GET, POST, PUT, DELETE**. REST APIs in Flask enable data exchange in JSON format between the backend and frontend or other applications.

#### Why Use Flask for API Development?

- Lightweight and easy to set up.
- Provides Flask-RESTful, a powerful extension for API development.
- Easily integrates with databases (e.g., MySQL, PostgreSQL).
- Supports authentication, validation, and error handling.

### Step-by-Step Implementation

Let's build a Flask REST API for a simple Task Management System that allows users to create, read, update, and delete (CRUD) tasks.

#### Step 1: Install Flask and Flask-RESTful

Before writing the code, install the necessary dependencies:

```
pip install flask flask-restful
```

## Step 2: Create a Flask App

Create a file called app.py and start writing the Flask API.

```
from flask import Flask  
from flask_restful import Api  
  
app = Flask(__name__)  
api = Api(app)
```

```
if __name__ == "__main__":  
    app.run(debug=True)
```

- We import Flask and Api from flask\_restful.
- Initialize the Flask app.
- Use Api(app) to define the API.
- Run the Flask app in debug mode.

## Step 3: Creating API Routes (GET, POST, PUT, DELETE)

Now, let's create CRUD operations for managing tasks.

Define a sample data structure

We will store tasks in a dictionary.

```
tasks = {  
    1: {"title": "Buy groceries", "completed": False},  
    2: {"title": "Read a book", "completed": True}  
}
```

**Step 4: Handling JSON Data with request.get\_json()**

To process user input as JSON, we use `request.get_json()`.

```
from flask import request  
from flask_restful import Resource
```

```
class Task(Resource):  
    def get(self, task_id):  
        """Retrieve a specific task."""  
        if task_id in tasks:  
            return tasks[task_id], 200  
        return {"message": "Task not found"}, 404  
  
    def post(self, task_id):  
        """Add a new task."""  
        if task_id in tasks:  
            return {"message": "Task ID already exists"}, 400  
        data = request.get_json()  
        tasks[task_id] = {"title": data["title"], "completed": data["completed"]}  
        return tasks[task_id], 201  
  
    def put(self, task_id):  
        """Update an existing task."""  
        if task_id not in tasks:  
            return {"message": "Task not found"}, 404  
        data = request.get_json()  
        tasks[task_id]["title"] = data["title"]  
        tasks[task_id]["completed"] = data["completed"]  
        return tasks[task_id], 200  
  
    def delete(self, task_id):
```

```
"""Delete a task."""
if task_id not in tasks:
    return {"message": "Task not found"}, 404
del tasks[task_id]
return {"message": "Task deleted"}, 200

• GET /tasks/<task_id> → Fetch task by ID.
• POST /tasks/<task_id> → Add a new task.
• PUT /tasks/<task_id> → Update task details.
• DELETE /tasks/<task_id> → Remove a task.
```

### **Step 5: Using Flask-RESTful to Build APIs**

Now, add the Task resource to the API.

```
api.add_resource(Task, "/tasks/<int:task_id>")
```

This connects our Task class to the /tasks/<task\_id> route.

### **Step 6: Testing APIs with Postman**

1. **Start the Flask app:** python app.py
  
2. **Open Postman** and test the endpoints:
  - a. GET /tasks/1 → Fetch task 1.
  - b. POST /tasks/3 → Add a new task.
  - c. PUT /tasks/1 → Update task 1.
  - d. DELETE /tasks/2 → Delete task 2.

### **Final app.py Code**

```
from flask import Flask, request
from flask_restful import Api, Resource
```

```
app = Flask(__name__)
api = Api(app)

tasks = {
    1: {"title": "Buy groceries", "completed": False},
    2: {"title": "Read a book", "completed": True}
}

class Task(Resource):
    def get(self, task_id):
        if task_id in tasks:
            return tasks[task_id], 200
        return {"message": "Task not found"}, 404

    def post(self, task_id):
        if task_id in tasks:
            return {"message": "Task ID already exists"}, 400
        data = request.get_json()
        tasks[task_id] = {"title": data["title"], "completed": data["completed"]}
        return tasks[task_id], 201

    def put(self, task_id):
        if task_id not in tasks:
            return {"message": "Task not found"}, 404
        data = request.get_json()
        tasks[task_id]["title"] = data["title"]
        tasks[task_id]["completed"] = data["completed"]
        return tasks[task_id], 200

    def delete(self, task_id):
        if task_id not in tasks:
```

```
        return {"message": "Task not found"}, 404
del tasks[task_id]
return {"message": "Task deleted"}, 200

api.add_resource(Task, "/tasks/<int:task_id>")

if __name__ == "__main__":
    app.run(debug=True)
```

## Conclusion

- We built a Flask REST API for managing tasks.
- Used Flask-RESTful to create API routes.
- Handled JSON data with request.get\_json().
- Tested the API using Postman.

## Mini Project 1: Online Bookstore API

### Project Overview

Create a RESTful API for an Online Bookstore, allowing users to view, add, update, and delete books. This API will handle book details like title, author, price, and availability.

### Step-by-Step Implementation

#### Step 1: Install Flask and Flask-RESTful

Run the following command to install the required packages:

```
pip install flask flask-restful
```

## Step 2: Create the Flask Application

Create a file app.py and initialize the Flask app:

```
from flask import Flask, request  
from flask_restful import Api, Resource  
  
app = Flask(__name__)  
api = Api(app)
```

## Step 3: Define the Data Structure

Store books in a dictionary:

```
books = {  
    1: {"title": "The Alchemist", "author": "Paulo Coelho", "price": 10.99,  
        "available": True},  
    2: {"title": "Atomic Habits", "author": "James Clear", "price": 15.50, "available":  
        True}  
}
```

## Step 4: Create the Book Resource

Define the API endpoints:

```
class Book(Resource):  
  
    def get(self, book_id):  
        """Retrieve book details by ID"""  
        if book_id in books:  
            return books[book_id], 200  
        return {"message": "Book not found"}, 404  
  
    def post(self, book_id):  
        """Add a new book"""  
        if book_id in books:
```

```

        return {"message": "Book ID already exists"}, 400
data = request.get_json()
books[book_id] = {
    "title": data["title"],
    "author": data["author"],
    "price": data["price"],
    "available": data["available"]
}
return books[book_id], 201

def put(self, book_id):
    """Update book details"""
    if book_id not in books:
        return {"message": "Book not found"}, 404
    data = request.get_json()
    books[book_id].update(data)
    return books[book_id], 200

def delete(self, book_id):
    """Delete a book"""
    if book_id not in books:
        return {"message": "Book not found"}, 404
    del books[book_id]
    return {"message": "Book deleted"}, 200

```

### **Step 5: Add Resource to API**

Connect the resource to the API:

```
api.add_resource(Book, "/books/<int:book_id>")
```

### **Step 6: Run the Flask App**

```
if __name__ == "__main__":
    app.run(debug=True)
```

### **Step 7: Test with Postman**

- GET /books/1 → Retrieve book details.
- POST /books/3 → Add a new book.
- PUT /books/1 → Update book details.
- DELETE /books/2 → Remove a book.

This API can be expanded by connecting to a database (MySQL, PostgreSQL) or adding authentication (JWT, OAuth2).

## **Mini Project 2: Student Management API**

### **Project Overview**

Develop a Student Management API to store and manage student records, including name, age, grade, and enrolled courses.

### **Step-by-Step Implementation**

#### **Step 1: Install Flask and Flask-RESTful**

pip install flask flask-restful

#### **Step 2: Create the Flask Application**

Create a new file student\_api.py and initialize Flask:

```
from flask import Flask, request
from flask_restful import Api, Resource
```

```
app = Flask(__name__)
api = Api(app)
```

### Step 3: Define the Data Structure

Store student records:

```
students = {
    101: {"name": "John Doe", "age": 20, "grade": "A", "courses": ["Math", "Physics"]},
    102: {"name": "Jane Smith", "age": 22, "grade": "B", "courses": ["Chemistry", "Biology"]}
}
```

### Step 4: Create the Student Resource

Define the API endpoints:

```
class Student(Resource):
    def get(self, student_id):
        """Retrieve student details by ID"""
        if student_id in students:
            return students[student_id], 200
        return {"message": "Student not found"}, 404
    def post(self, student_id):
        """Add a new student"""
        if student_id in students:
            return {"message": "Student ID already exists"}, 400
        data = request.get_json()
        students[student_id] = {
            "name": data["name"],
            "age": data["age"],
            "grade": data["grade"],
            "courses": data["courses"]
        }
```

```

return students[student_id], 201

def put(self, student_id):
    """Update student details"""
    if student_id not in students:
        return {"message": "Student not found"}, 404
    data = request.get_json()
    students[student_id].update(data)
    return students[student_id], 200

def delete(self, student_id):
    """Delete a student"""
    if student_id not in students:
        return {"message": "Student not found"}, 404
    del students[student_id]
    return {"message": "Student deleted"}, 200

```

**Step 5: Add Resource to API**

```
api.add_resource(Student, "/students/<int:student_id>")
```

**Step 6: Run the Flask App**

```
if __name__ == "__main__":
    app.run(debug=True)
```

**Step 7: Test with Postman**

- GET /students/101 → Retrieve student details.
- POST /students/103 → Add a new student.
- PUT /students/101 → Update student details.
- DELETE /students/102 → Remove a student.

This API can be extended by:

- Connecting to a database (MySQL, PostgreSQL).
- Implementing authentication (JWT, OAuth2).
- Adding pagination and search functionalities.

## Conclusion

Both projects demonstrate: ✓ Building RESTful APIs using Flask & Flask-RESTful.

- ✓ Creating API routes (GET, POST, PUT, DELETE).
- ✓ Handling JSON data using `request.get_json()`.
- ✓ Testing APIs using Postman.

## Mini Project: Employee Management System API

### Project Overview

Develop a RESTful API for an Employee Management System, allowing users to add, update, retrieve, and delete employee records. The API will handle employee details such as name, age, department, salary, and employment status.

### Day 81 Tasks

#### Task 1: Setup Flask and Flask-RESTful

- Install required packages.
- Initialize the Flask app.

#### Task 2: Create a basic Flask application

- Create a new Python file `app.py`.
- Define a simple Flask route to test API functionality.

Task 3: Define the employee data structure

- Use a dictionary to store employee details with unique IDs.

Task 4: Create an API resource for employee records

- Define an Employee resource class using Flask-RESTful.

Task 5: Implement a GET request to retrieve employee details

- Fetch details of a specific employee using an employee ID.

Task 6: Implement a POST request to add a new employee

- Allow users to send JSON data to add a new employee.

Task 7: Implement a PUT request to update employee details

- Modify existing employee records using an employee ID.

Task 8: Implement a DELETE request to remove an employee

- Delete an employee record by providing an employee ID.

Task 9: Handle missing employee records

- Return proper error messages (404 Not Found, 400 Bad Request).

Task 10: Validate JSON input data

- Ensure valid employee details are received before processing.

**Task 11: Implement status-based filtering**

- Allow users to filter employees based on their employment status (Active/Inactive).

**Task 12: Implement an endpoint to retrieve all employees**

- Provide an API endpoint to fetch all employee records.

**Task 13: Test the API using Postman**

- Send GET, POST, PUT, DELETE requests using Postman.
- Verify responses and debug errors.

This Employee Management System API will demonstrate:

- ✓ Building RESTful APIs using Flask & Flask-RESTful
- ✓ CRUD operations (Create, Read, Update, Delete)
- ✓ Handling JSON data (`request.get_json()`)
- ✓ Validating and filtering API data
- ✓ Testing with Postman

**Mini Project 1: Employee Management API**

**Project Requirements**

Develop a RESTful API for an Employee Management System where users (HR/Admin) can manage employee records. The API should include the following functionalities:

## 1. Employee Management

- a. Add a new employee with details (name, department, email, salary, joining date).
- b. Retrieve an employee's details by ID.
- c. Update an employee's information.
- d. Delete an employee from the system.
- e. Retrieve all employees.

## 2. Department Management

- a. Add, update, and delete departments.
- b. Retrieve all employees under a specific department.

## 3. Filtering & Sorting

- a. Search employees by name or department.
- b. Filter employees based on salary range or joining date.

## 4. API Testing

- a. Use Postman to test API routes for GET, POST, PUT, and DELETE requests.

## Mini Project 2: Task Management API

### Project Requirements

Develop a Task Management API that allows users to manage tasks efficiently.

The API should support:

## 1. Task Operations

- a. Create a new task with a title, description, deadline, and status (Pending, In Progress, Completed).
- b. Retrieve task details by ID.
- c. Update task information (e.g., change status or deadline).
- d. Delete a task.
- e. Retrieve all tasks.

## 2. Filtering & Sorting

- a. Filter tasks based on status (Pending, In Progress, Completed).
- b. Sort tasks by deadline.

## 3. User Management (Optional)

- a. Assign tasks to specific users.
- b. Retrieve tasks assigned to a user.

## 4. API Testing

- a. Test all API routes using **Postman**.

# Day 82

## Flask & Frontend Integration (AJAX, Fetch API, JavaScript, Bootstrap)

### 1. Introduction

Flask is a powerful backend framework that allows us to build web applications and APIs. However, to create dynamic, interactive, and user-friendly applications, we must integrate it with frontend technologies like AJAX, Fetch API, JavaScript, and Bootstrap.

This integration helps us:

- Send and receive data between the backend and frontend without reloading the page.
- Improve the user experience by displaying real-time updates.
- Style Flask apps easily using Bootstrap.
- Use Jinja2 templating with JavaScript to build interactive and data-driven web applications.

## 2. Key Concepts with Step-by-Step Implementation

### Serving JSON Data to the Frontend

Flask allows us to send JSON responses to the frontend. JSON (JavaScript Object Notation) is the standard format for data exchange between the frontend and backend.

### Syntax for Returning JSON in Flask

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/data', methods=['GET'])
def send_json():
    data = {"name": "John Doe", "age": 30, "city": "New York"}
    return jsonify(data) # Convert Python dictionary to JSON

if __name__ == '__main__':
    app.run(debug=True)
```

### Step-by-Step Implementation

1. Import Flask and jsonify.
2. Create an API endpoint (/data) that returns a JSON response.
3. Run the Flask app and visit <http://127.0.0.1:5000/data> in your browser.
4. You should see the JSON output: {"name": "John Doe", "age": 30, "city": "New York"}

## Fetching API Data with JavaScript (AJAX, Fetch API)

JavaScript can fetch data from the Flask backend without refreshing the page using the Fetch API or AJAX (jQuery).

### Fetch API Example (JavaScript)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Fetch API Example</title>
    <script>
        function fetchData() {
            fetch('/data') // Request JSON data from Flask
                .then(response => response.json()) // Convert response to JSON
                .then(data => {
                    document.getElementById("result").innerHTML =
                        `Name: ${data.name}, Age: ${data.age}, City: ${data.city}`;
                })
                .catch(error => console.error("Error fetching data:", error));
        }
    </script>
</head>
<body>
    <h2>Fetch API Example</h2>
    <button onclick="fetchData()">Get Data</button>
    <p id="result"></p>
</body>
</html>
```

### Step-by-Step Implementation

1. Create an HTML file.

2. Add a button that, when clicked, fetches data from /data.
3. Use JavaScript's fetch() method to call the Flask API.
4. Display the JSON response dynamically on the webpage.

## Using Bootstrap for Styling Flask Apps

Bootstrap is a popular CSS framework that helps in designing responsive and visually appealing web applications.

### Flask App with Bootstrap

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Bootstrap Example</title>
  <link rel="stylesheet"
  href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css"
  >
</head>
<body>
  <div class="container">
    <h1 class="text-center text-primary">Welcome to Flask + Bootstrap</h1>
    <button class="btn btn-success" onclick="fetchData()">Load Data</button>
    <p id="result" class="alert alert-info mt-3"></p>
  </div>
</body>
</html>
```

### Step-by-Step Implementation

1. Include Bootstrap via CDN in the <head>.
2. Use Bootstrap classes like container, btn, text-primary, etc.
3. Style buttons, text, and alerts dynamically.

## Building Dynamic Flask Apps with Jinja2 + JavaScript

Jinja2 allows us to pass data from Flask to the frontend dynamically. We can combine it with JavaScript to create interactive applications.

### Flask Code (app.py)

```
from flask import Flask, render_template
```

```
app = Flask(__name__)
@app.route('/')
def index():
    users = ["Alice", "Bob", "Charlie"]
    return render_template('index.html', users=users)
if __name__ == '__main__':
    app.run(debug=True)
```

### HTML Code (index.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Jinja2 + JavaScript</title>
</head>
<body>
    <h2>User List</h2>
    <ul>
        {% for user in users %}
            <li>{{ user }}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

## Step-by-Step Implementation

1. Create a Flask route (/) that passes a list of users to the template.
2. Use Jinja2 {% for user in users %} to loop through and display names.
3. Load the page and see dynamically rendered content.

## 3. Conclusion

Flask can be seamlessly integrated with frontend technologies for real-time, dynamic, and visually appealing web applications. By using:

- ✓ JSON responses → We send structured data to the frontend.
- ✓ Fetch API/AJAX → We update the UI without reloading.
- ✓ Bootstrap → We style applications responsively.
- ✓ Jinja2 & JavaScript → We create dynamic and interactive experiences.

## Mini Project 1: Real-Time Weather App Using Flask & Fetch API

### Project Overview

This project allows users to enter a city name and fetch real-time weather data from an API using Flask and JavaScript's Fetch API.

### Features

- ✓ User enters a city name.
- ✓ Flask fetches weather data from an external API (OpenWeatherMap).
- ✓ Data is sent to the frontend as JSON.
- ✓ JavaScript updates the UI dynamically without reloading.
- ✓ Bootstrap is used for styling.

## Step 1: Install Flask & Required Packages

Run this command in your terminal:

```
pip install flask requests
```

## Step 2: Create Flask Backend (app.py)

```
from flask import Flask, render_template, request, jsonify
import requests
```

```
app = Flask(__name__)
# OpenWeatherMap API Key (Replace with your own)
API_KEY = "your_api_key_here"
@app.route('/')
def index():
    return render_template('index.html')

@app.route('/get_weather', methods=['POST'])
def get_weather():
    data = request.get_json()
    city = data.get("city")
    if not city:
        return jsonify({"error": "City name is required"}), 400

    # Fetch weather data from OpenWeatherMap API
    url =
    f"http://api.openweathermap.org/data/2.5/weather?q={city}&appid={API_KEY}&
units=metric"
    response = requests.get(url).json()
    if response.get("cod") != 200:
        return jsonify({"error": "City not found"}), 404
```

```

# Extract relevant data
weather_info = {
    "city": response["name"],
    "temperature": response["main"]["temp"],
    "description": response["weather"][0]["description"]
}
return jsonify(weather_info)
if __name__ == '__main__':
    app.run(debug=True)

```

**Explanation:**

- index.html is loaded when the user visits /.
- /get\_weather receives city name from the frontend via AJAX (Fetch API).
- Requests weather data from OpenWeatherMap API.
- Returns the weather as JSON to the frontend.

**Step 3: Create Frontend (templates/index.html)**

```

<!DOCTYPE html>
<html lang="en">
<head>
    <title>Weather App</title>
    <link rel="stylesheet"
        href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css"
    >
    <script>
        function fetchWeather() {
            let city = document.getElementById("city").value;
            fetch('/get_weather', {
                method: "POST",
                headers: { "Content-Type": "application/json" },
                body: JSON.stringify({ city: city })

```

```

        })
        .then(response => response.json())
        .then(data => {
            if (data.error) {
                document.getElementById("weather-info").innerHTML = `<div
class="alert alert-danger">${data.error}</div>`;
            } else {
                document.getElementById("weather-info").innerHTML = `
                    <h3>${data.city}</h3>
                    <p>Temperature: ${data.temperature}°C</p>
                    <p>Condition: ${data.description}</p>
                `;
            }
        })
        .catch(error => console.error("Error:", error));
    }

```

</script>

</head>

<body class="container mt-5">

<h2 class="text-center">Weather App</h2>

<div class="mb-3">

<input type="text" id="city" class="form-control" placeholder="Enter city name">

<button class="btn btn-primary mt-2" onclick="fetchWeather()">Get Weather</button>

</div>

<div id="weather-info" class="mt-3"></div>

</body>

</html>

### **Explanation:**

- User enters a city name and clicks "Get Weather".
- JavaScript sends an AJAX request using Fetch API to /get\_weather.
- Response is displayed dynamically on the webpage without reloading.
- Bootstrap is used for styling.

### **Step 4: Run the Application**

python app.py

Open <http://127.0.0.1:5000/> and test the **Weather App!**

## **Mini Project 2: To-Do List Using Flask, Jinja2 & AJAX**

### **Project Overview**

This project allows users to add and remove tasks in a to-do list dynamically using Flask, Jinja2, and AJAX.

### **Features**

- ✓ Add tasks dynamically.
- ✓ Remove tasks instantly without refreshing.
- ✓ Flask backend stores tasks in memory.
- ✓ Jinja2 renders the task list.
- ✓ Bootstrap is used for styling.

### **Step 1: Install Flask**

Run this command:

pip install flask

**Step 2: Create Flask Backend (app.py)**

```
from flask import Flask, render_template, request, jsonify

app = Flask(__name__)

tasks = [] # In-memory task list

@app.route('/')
def index():
    return render_template('index.html', tasks=tasks)

@app.route('/add_task', methods=['POST'])
def add_task():
    data = request.get_json()
    task = data.get("task")

    if not task:
        return jsonify({"error": "Task cannot be empty"}), 400

    tasks.append(task)
    return jsonify({"message": "Task added successfully", "tasks": tasks})

@app.route('/delete_task', methods=['POST'])
def delete_task():
    data = request.get_json()
    task = data.get("task")

    if task in tasks:
        tasks.remove(task)
        return jsonify({"message": "Task removed", "tasks": tasks})
```

```
return jsonify({"error": "Task not found"}), 404
```

```
if __name__ == '__main__':
    app.run(debug=True)
```

**Explanation:**

- The backend has two AJAX routes:
  - /add\_task → Adds a task.
  - /delete\_task → Removes a task.
- Tasks are stored in a Python **list** (tasks[]).

**Step 3: Create Frontend (templates/index.html)**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>To-Do List</title>
  <link rel="stylesheet"
  href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css"
>
<script>
  function addTask() {
    let task = document.getElementById("task").value;

    fetch('/add_task', {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ task: task })
    })
    .then(response => response.json())
    .then(data => {
      if (data.error) {
```

```

        alert(data.error);
    } else {
        updateTaskList(data.tasks);
    }
});

}

function deleteTask(task) {
    fetch('/delete_task', {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ task: task })
    })
    .then(response => response.json())
    .then(data => updateTaskList(data.tasks));
}

function updateTaskList(tasks) {
    let taskList = document.getElementById("task-list");
    taskList.innerHTML = "";
    tasks.forEach(task => {
        taskList.innerHTML += `<li class="list-group-item">
            ${task} <button class="btn btn-danger btn-sm float-end"
            onclick="deleteTask('${task}')">X</button>
        </li>`;
    });
}
</script>
</head>
<body class="container mt-5">
    <h2 class="text-center">To-Do List</h2>

```

```
<input type="text" id="task" class="form-control" placeholder="Enter task">
<button class="btn btn-primary mt-2" onclick="addTask()">Add Task</button>
<ul id="task-list" class="list-group mt-3"></ul>
</body>
</html>
```

#### **Step 4: Run the Application**

python app.py

Open <http://127.0.0.1:5000/> and test the **To-Do List!**

### **Real-Life Mini Project: Student Management System**

This Student Management System project will allow users to add, view, update, and delete student records dynamically using Flask, JavaScript (AJAX, Fetch API), Bootstrap, and Jinja2.

#### **Day 82 Tasks**

##### **Step 1: Set Up Flask Project Structure**

- Create a Flask project structure with app.py, static/, and templates/.
- Install required dependencies (flask).

##### **Step 2: Create Flask Backend (app.py)**

- Set up the Flask application.
- Create a route to serve the HTML template (index.html).

##### **Step 3: Design the Frontend with Bootstrap (index.html)**

- Use Bootstrap to create a table for displaying student records.
- Add an input form to add new students dynamically.

#### Step 4: Implement JSON API Route to Get Student Data

- Create a Flask route (/students) that returns JSON student data.

#### Step 5: Fetch Student Data Using JavaScript (AJAX/Fetch API)

- Use Fetch API to load student data dynamically when the page loads.
- Display student data inside the Bootstrap table without page reload.

#### Step 6: Implement Add Student Functionality

- Create a route (/add\_student) to add a student via an AJAX POST request.
- Update the frontend dynamically after adding a student.

#### Step 7: Implement Delete Student Functionality

- Create a route (/delete\_student) to delete a student via an AJAX DELETE request.
- Remove the deleted student dynamically from the UI.

#### Step 8: Implement Edit Student Functionality

- Add an "Edit" button for each student in the table.
- Implement an AJAX request (/update\_student) to edit student details.

#### Step 9: Build a Modal Popup for Editing Student Details

- Use Bootstrap Modals to show a popup form when editing a student.

#### Step 10: Add Real-Time Search Functionality

- Implement a search bar that filters student records dynamically.

### Step 11: Display Flash Messages for User Actions

- Show success/error messages when adding, editing, or deleting students.

### Step 12: Implement Session Management

- Use Flask sessions to keep track of user interactions.

### Step 13: Deploy the Application

- Host the project using Flask's built-in server or deploy on Heroku/Hostinger.

### Outcome

By completing this project, you will build a fully dynamic Student Management System that integrates Flask and JavaScript to handle data efficiently without requiring page reloads.

## Real-Life Mini Project 1: Employee Attendance Tracker

### Description:

Build a web-based Employee Attendance Tracker where employees can mark their attendance, and admins can view, update, and delete records dynamically using Flask, JavaScript (AJAX, Fetch API), Bootstrap, and Jinja2.

### Features:

- Employees can mark their daily attendance (Check-in & Check-out).
- Admins can view attendance records dynamically.
- Implement AJAX-based CRUD operations (Add, Edit, Delete attendance records).
- Display attendance logs in a Bootstrap-styled table.
- Use Bootstrap modals for editing records.
- Implement a search filter to find attendance records quickly.

## Real-Life Mini Project 2: Real-Time Stock Price Dashboard

### Description:

Create a Real-Time Stock Price Dashboard that fetches stock data from an external API and displays it dynamically using Flask, JavaScript (Fetch API), Bootstrap, and Jinja2.

### Features:

- Fetch live stock price data using an API (e.g., Yahoo Finance or Alpha Vantage).
- Display stock details in a Bootstrap-styled dashboard.
- Implement AJAX auto-refresh to update stock prices dynamically.
- Allow users to search for stock symbols and fetch real-time data.
- Use charts/graphs (Chart.js or D3.js) to visualize stock trends dynamically.
- Implement Flash messages for API errors (e.g., invalid stock symbol).

## Day 83

### Deploying Flask Applications

When you build a Flask application, you need to deploy it so users can access it over the internet. Running Flask locally is good for development, but in production, we need to:

- ✓ Use Gunicorn or uWSGI to run the application efficiently.
- ✓ Deploy on cloud platforms like Heroku, AWS, or PythonAnywhere.
- ✓ Use Docker to containerize the application for easy deployment.
- ✓ Manage environment variables for security.

## 1. Preparing Flask for Production (Gunicorn, uWSGI)

Flask's default development server is not suitable for production because:

- It handles only one request at a time.
- It lacks security and error handling.
- It crashes when handling large requests.

That's why we use Gunicorn and uWSGI, which are WSGI-compliant web servers.

### Step 1: Install Gunicorn & uWSGI

```
pip install gunicorn uwsgi
```

### Step 2: Running Flask with Gunicorn

Let's say you have a simple Flask app (app.py):

```
from flask import Flask
```

```
app = Flask(__name__)
@app.route('/')
def home():
    return "Hello, Flask in Production!"
if __name__ == '__main__':
    app.run()
```

Run it with Gunicorn:

```
gunicorn -w 4 -b 0.0.0.0:8000 app:app
```

### Explanation

- `-w 4` → Uses 4 worker processes to handle multiple requests.
- `-b 0.0.0.0:8000` → Runs the app on port 8000.
- `app:app` → Refers to Flask file (app.py) and Flask app object (app).

### Step 3: Running Flask with uWSGI

Another option is uWSGI, which also runs Flask in production.

```
uwsgi --http :8000 --wsgi-file app.py --callable app --processes 4 --threads 2
```

#### Explanation

- --processes 4 → Runs 4 processes.
- --threads 2 → Each process handles 2 threads.
- --wsgi-file app.py --callable app → Runs the Flask app.

Now, your Flask app can handle multiple requests efficiently!

## 2. Deploying Flask Apps on Heroku, AWS, or PythonAnywhere

Once the app is production-ready, we need to deploy it.

### Option 1: Deploying on Heroku

Heroku is a free cloud platform for hosting Flask apps.

#### Step 1: Install Heroku CLI

```
curl https://cli-assets.heroku.com/install.sh | sh
```

#### Step 2: Login & Create Heroku App

```
heroku login
```

```
heroku create flask-app-example
```

#### Step 3: Add a Procfile for Gunicorn

Create a Procfile and add:

```
web: gunicorn app:app
```

#### **Step 4: Deploy to Heroku**

```
git init  
git add .  
git commit -m "Deploy to Heroku"  
heroku git:remote -a flask-app-example  
git push heroku master
```

Your Flask app is now live on Heroku!

#### **Option 2: Deploying on PythonAnywhere**

PythonAnywhere is a cloud platform designed for Python applications.

Steps to Deploy

1. Create an account at [PythonAnywhere](#).
2. Upload your Flask files via the file manager.
3. Create a virtual environment: mkvirtualenv myflaskenv --python=python3.9  
pip install flask gunicorn
4. Set up WSGI configuration in the dashboard.
5. Run the app and get a public URL.

Now your Flask app is live on PythonAnywhere!

### **3. Using Docker for Flask Applications**

Docker packages your Flask app so it can run anywhere.

#### **Step 1: Install Docker**

If you haven't installed Docker, get it from: [Docker Official Website](#)

#### **Step 2: Create a Dockerfile**

Inside your Flask project folder, create a file called Dockerfile:

```
# Use Python as the base image
FROM python:3.9

# Set the working directory
WORKDIR /app

# Copy project files into the container
COPY ..

# Install dependencies
RUN pip install -r requirements.txt

# Run the application
CMD ["gunicorn", "-w", "4", "-b", "0.0.0.0:8000", "app:app"]
```

### **Step 3: Build & Run the Docker Container**

```
docker build -t flask-app .
docker run -p 8000:8000 flask-app
```

Now your Flask app is running inside Docker!

## **4. Environment Variables & Configuration Management**

Storing sensitive data inside code is dangerous. Instead, use environment variables.

### **Step 1: Install python-dotenv**

```
pip install python-dotenv
```

## Step 2: Create a .env File

Inside your project, create a .env file:

```
SECRET_KEY=mysecretkey  
DATABASE_URL=mysql://user:password@localhost/dbname
```

## Step 3: Load .env Variables in Flask

Modify app.py:

```
from flask import Flask  
from dotenv import load_dotenv  
import os  
  
# Load environment variables  
load_dotenv()  
  
app = Flask(__name__)  
app.config['SECRET_KEY'] = os.getenv('SECRET_KEY')  
  
@app.route('/')  
def home():  
    return "Flask App with Environment Variables!"  
  
if __name__ == '__main__':  
    app.run(debug=True)
```

Now, Flask securely loads environment variables from .env!

## Summary of Key Topics

Feature	Description
Gunicorn & uWSGI	Run Flask in production mode with multiple worker processes.
Deploy on Heroku	Host your Flask app for free using Git & Gunicorn.
Deploy on PythonAnywhere	Simple way to host Flask apps with minimal setup.
Docker for Flask	Use Docker containers to deploy Flask anywhere.
Environment Variables	Secure sensitive data using .env files and os.getenv().

## Day 84

We will build a full Flask eCommerce project step by step, covering everything from setup to deployment. This project will include:

- ✓ User authentication (signup, login, logout)
- ✓ Product management (add, update, delete, list products)
- ✓ Shopping cart functionality
- ✓ REST API integration
- ✓ AJAX & Fetch API for frontend interactions
- ✓ Database operations with Flask-SQLAlchemy
- ✓ Deployment of the project

## Step 1: Project Setup

### 1. Install Required Packages

Before starting, install Flask and other dependencies:

```
pip install flask flask-sqlalchemy flask-login flask-restful flask-wtf werkzeug
```

### 2. Create the Project Structure

```
ecommerce_project/
```

```
|—— app.py  
|—— config.py  
|—— database.db  
|—— static/  
|   |—— css/  
|   |—— js/  
|—— templates/  
|   |—— base.html  
|   |—— home.html  
|   |—— login.html  
|   |—— register.html  
|   |—— products.html  
|—— models.py  
|—— forms.py  
|—— routes.py  
|—— api.py  
|—— requirements.txt
```

## Step 2: Flask App & Configuration

### 1. Create app.py

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_login import LoginManager

app = Flask(__name__)
app.config.from_object('config')

db = SQLAlchemy(app)
login_manager = LoginManager(app)

from routes import *
from api import *

if __name__ == '__main__':
    app.run(debug=True)
```

### 2. Configure Database in config.py

```
import os

BASE_DIR = os.path.abspath(os.path.dirname(__file__))

class Config:
    SECRET_KEY = 'your_secret_key'
    SQLALCHEMY_DATABASE_URI = 'sqlite:///{} + os.path.join(BASE_DIR,
'database.db')
    SQLALCHEMY_TRACK_MODIFICATIONS = False
```

### Step 3: Database Models

#### 1. Define User & Product Models in models.py

```
from app import db
from flask_login import UserMixin

class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(100), unique=True, nullable=False)
    email = db.Column(db.String(100), unique=True, nullable=False)
    password = db.Column(db.String(100), nullable=False)

class Product(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100), nullable=False)
    price = db.Column(db.Float, nullable=False)
    description = db.Column(db.Text, nullable=False)
```

#### 2. Initialize the Database

Run the following in Python shell:

```
python
>>> from app import db
>>> db.create_all()
```

### Step 4: User Authentication (Signup, Login, Logout)

#### 1. Create Flask-WTF Forms in forms.py

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, SubmitField
from wtforms.validators import DataRequired, Email, EqualTo
```

```
class RegisterForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
    email = StringField('Email', validators=[DataRequired(), Email()])
    password = PasswordField('Password', validators=[DataRequired()])
    confirm_password = PasswordField('Confirm Password',
                                    validators=[DataRequired(), EqualTo('password')])
    submit = SubmitField('Register')
```

## 2. Implement Authentication Routes in routes.py

```
from flask import render_template, redirect, url_for, request, flash
from app import app, db
from models import User
from forms import RegisterForm
from werkzeug.security import generate_password_hash, check_password_hash
from flask_login import login_user, logout_user

@app.route('/register', methods=['GET', 'POST'])
def register():
    form = RegisterForm()
    if form.validate_on_submit():
        hashed_password = generate_password_hash(form.password.data,
                                                method='pbkdf2:sha256')
        new_user = User(username=form.username.data, email=form.email.data,
                        password=hashed_password)
        db.session.add(new_user)
        db.session.commit()
        flash('Account created successfully!', 'success')
        return redirect(url_for('login'))
    return render_template('register.html', form=form)
```

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    return render_template('login.html')
```

## **Step 5: Product Management**

### **1. Create Routes to List & Add Products**

```
@app.route('/products', methods=['GET', 'POST'])
def products():
    if request.method == 'POST':
        name = request.form['name']
        price = request.form['price']
        description = request.form['description']
        new_product = Product(name=name, price=price, description=description)
        db.session.add(new_product)
        db.session.commit()
    products = Product.query.all()
    return render_template('products.html', products=products)
```

### **2. Create products.html**

```
{% for product in products %}
<div>
    <h3>{{ product.name }}</h3>
    <p>Price: ${{ product.price }}</p>
    <p>{{ product.description }}</p>
</div>
{% endfor %}
```

## Step 6: REST API for Products

### 1. Create api.py

```
from flask_restful import Resource, Api
from app import app, db
from models import Product
api = Api(app)

class ProductAPI(Resource):
    def get(self, product_id):
        product = Product.query.get_or_404(product_id)
        return {"id": product.id, "name": product.name, "price": product.price,
                "description": product.description}

api.add_resource(ProductAPI, '/api/product/<int:product_id>')
```

## Step 7: Fetching API Data in Frontend

### 1. Using Fetch API in JavaScript

```
fetch('/api/product/1')
    .then(response => response.json())
    .then(data => {
        console.log("Product Name:", data.name);
        console.log("Price:", data.price);
    });
});
```

## Step 8: Deployment

### 1. Prepare Flask for Production

pip install gunicorn

Modify app.py:

```
if __name__ == "__main__":
    app.run(host='0.0.0.0', port=5000)
```

## 2. Deploy on PythonAnywhere

1. Create a free account on PythonAnywhere.
2. Upload your project files.
3. Set up a WSGI application pointing to app.py.
4. Configure SQLite/MySQL as needed.

## Final eCommerce App Summary

Feature	Implementation
User Authentication	Flask-WTF forms, login, register
Database	Flask-SQLAlchemy, SQLite/MySQL
Product Management	Add, Update, Delete, List products
REST API	Flask-RESTful for product API
Frontend Integration	Fetch API & JavaScript for dynamic UI
Deployment	PythonAnywhere with Gunicorn

## Power BI

# Day 85

### Introduction to Power BI & Setup

Power BI is a powerful business intelligence tool developed by Microsoft. It allows users to connect, transform, analyze, and visualize data from various sources. With Power BI, users can create interactive dashboards and reports, making it easier to gain insights from their data.

## What is Power BI? Why use it?

### Definition:

Power BI is a data visualization and business intelligence tool that enables users to connect multiple data sources, clean and transform data, perform analysis, and create interactive reports and dashboards.

### Why use Power BI?

- ✓ Easy to use – No need for complex coding.
- ✓ Data Connectivity – Connects to multiple data sources like Excel, SQL, Google Sheets, APIs, etc.
- ✓ Data Transformation – Clean, shape, and model data easily using Power Query.
- ✓ Advanced Visualizations – Rich graphical representation with charts, graphs, and KPI indicators.
- ✓ Cloud Integration – Share reports and dashboards through Power BI Service.
- ✓ AI-powered Insights – Uses AI for smart analysis.

### Installing Power BI Desktop

Power BI Desktop is a free application that allows you to develop reports before publishing them online.

#### Step-by-Step Installation Guide

##### 1. Download Power BI Desktop

- a. Visit the official Microsoft Power BI website:  
<https://powerbi.microsoft.com/>
- b. Click on “Download Free” under Power BI Desktop.

## 2. Install Power BI Desktop

- a. Open the downloaded .exe file.
- b. Follow the installation wizard (Click **Next → Next → Finish**).
- c. Open Power BI Desktop after installation.

## Understanding Power BI Components

Power BI consists of three main components:

Component	Description
Power BI Desktop	Used for designing reports and data transformation.
Power BI Service	Cloud-based service for sharing reports and dashboards.
Power BI Mobile	Mobile app to access dashboards from anywhere.
Power BI Gateway	Connects on-premises data sources with Power BI Service.

## Exploring the Power BI Interface

When you open Power BI Desktop, you will see:

1. **Ribbon** – Contains buttons for importing data, transforming data, creating visuals, etc.
2. **Data Pane** – Displays all imported tables and fields.
3. **Report View** – The main workspace for designing dashboards.
4. **Fields Pane** – Shows available datasets for analysis.
5. **Visualizations Pane** – Drag-and-drop tools to create charts, graphs, and reports.

## Importing Basic Datasets (Excel, CSV)

Example: Import an Excel File into Power BI

**Scenario:** You have an Excel file (`sales_data.xlsx`) with sales records, and you want to analyze it in Power BI.

## Step-by-Step Implementation

1. Open Power BI Desktop
2. Click on “Get Data” → Select “Excel”
3. Browse and Open sales\_data.xlsx
4. Select the sheet (e.g., “SalesData”) and Click “Load”
5. Data is now imported into Power BI
6. Drag fields (e.g., Sales, Date, Product) into the visualization pane to create reports.

### For CSV files:

- Follow the same steps, but instead of Excel, choose CSV in the Get Data option.

### Real-Life Example: Creating a Sales Dashboard

Let's say you run an online store and want to analyze your monthly sales.

Steps:

1. Import sales data (sales\_data.xlsx)
2. Clean and transform data using Power Query
3. Create a bar chart to show sales trends over months
4. Add slicers to filter data by category or product
5. Publish and share the report via Power BI Service

### Summary

- ✓ Power BI is a powerful tool for data analysis and visualization.
- ✓ You can import and transform data from Excel, CSV, and other sources.
- ✓ Power BI Desktop is used to design reports, while Power BI Service is for cloud sharing.
- ✓ Real-life dashboards help businesses track performance effectively.

## Real-Life Mini Projects on Power BI

Here are two real-life Power BI mini projects with step-by-step implementation, including dataset requirements, step-by-step procedures, and detailed explanations.

### Mini Project 1: Sales Performance Dashboard

#### Project Requirement:

A retail company wants to analyze its monthly sales performance. The management team wants a Power BI dashboard to track total sales, top-selling products, and revenue trends.

#### Step 1: Prepare the Dataset

We'll use an Excel file (Sales\_Data.xlsx) with the following columns:

Order ID	Date	Product	Category	Region	Sales (\$)	Quantity
1001	01-01-2024	Laptop	Electronics	North	1500	2
1002	02-01-2024	Phone	Electronics	South	800	1
1003	03-01-2024	Shirt	Clothing	East	40	3

#### Step 2: Install & Open Power BI Desktop

- Download & Install Power BI Desktop from [Microsoft's Official Site](#).
- Open Power BI Desktop after installation.

#### Step 3: Import Dataset into Power BI

1. Click "Home" → "Get Data" → Select "Excel"
2. Browse & Open Sales\_Data.xlsx.
3. Select the worksheet (e.g., SalesData).
4. Click "Load" to import the dataset into Power BI.

#### Step 4: Exploring the Power BI Interface

Now, let's understand the Power BI interface:

- Data Pane → Displays imported datasets.
- Fields Pane → Lists all the columns from your dataset.
- Report View → Allows creating visualizations.

## **Step 5: Creating Sales Performance Dashboard**

Now, let's create three visualizations:

### **(1) Total Sales Calculation**

- Go to "Report View"
- Click "Card" visualization (from the Visualizations Pane)
- Drag "Sales (\$)" field into the "Values" section
- It will display Total Sales in a card format.

### **(2) Sales Trend (Line Chart)**

- Click "Line Chart" from the Visualizations Pane.
- Drag "Date" into the "X-axis" and "Sales (\$)" into the "Y-axis".
- Result: A sales trend chart showing revenue over time.

### **(3) Top-Selling Products (Bar Chart)**

- Click "Bar Chart" visualization.
- Drag "Product" into "Axis" and "Sales (\$)" into "Values".
- Sort in Descending Order to highlight the best-selling products.

## **Step 6: Publish & Share Report**

1. Click "File" → "Save" the report.
2. Click "Publish" to upload the report to Power BI Service.
3. Share the link with stakeholders.

## Final Output:

- ✓ A Sales Dashboard with Total Sales, Sales Trend, and Top-Selling Products.
- ✓ Managers can analyze performance monthly & region-wise.

## Mini Project 2: Customer Feedback Analysis Dashboard

### Project Requirement:

A company collects customer feedback from an online survey (CSV file). They want a Power BI dashboard to analyze customer satisfaction scores and common complaints.

### Step 1: Prepare the Dataset

We'll use a CSV file (Customer\_Feedback.csv) with these columns:

Customer ID	Date	Rating (1-5)	Feedback	Region
101	01-02-2024	5	Excellent service!	North
102	02-02-2024	2	Slow delivery	South
103	03-02-2024	3	Average quality	East

### Step 2: Install & Open Power BI Desktop

- If not already installed, download and install Power BI Desktop.
- Open Power BI Desktop.

### Step 3: Import Dataset into Power BI

1. Click "Home" → "Get Data" → Select "CSV"
2. Browse & Open Customer\_Feedback.csv
3. Click "Load" to import data.

#### **Step 4: Exploring the Power BI Interface**

- Fields Pane: Lists Customer ID, Date, Rating, Feedback, Region.
- Report View: Area where we create visualizations.

#### **Step 5: Creating Customer Feedback Dashboard**

Now, we will create two visualizations:

##### **(1) Average Customer Rating (Gauge Chart)**

- Click "Gauge Chart" visualization.
- Drag "Rating (1-5)" to "Values".
- Set Minimum to 1 and Maximum to 5.
- Result: A gauge showing average customer satisfaction.

##### **(2) Customer Ratings by Region (Column Chart)**

- Click "Clustered Column Chart" visualization.
- Drag "Region" into "Axis" and "Rating (1-5)" into "Values".
- Result: A bar chart showing average ratings per region.

#### **Step 6: Publish & Share Report**

1. Click “File” → “Save” the report.
2. Click “Publish” to upload the report to Power BI Service.
3. Share the link with the customer service team.

#### **Final Output:**

- ✓ A Customer Feedback Dashboard with Average Rating & Region-wise Analysis.
- ✓ Helps the company identify & improve low-rated areas.

## Summary

### Project 1: Sales Performance Dashboard

- ✓ Dataset: Sales records (Excel)
- ✓ Visuals: Total Sales (Card), Sales Trend (Line Chart), Top Products (Bar Chart)
- ✓ Business Use: Track revenue, sales trends, and product performance.

### Project 2: Customer Feedback Dashboard

- ✓ Dataset: Customer feedback (CSV)
- ✓ Visuals: Average Rating (Gauge Chart), Region-wise Ratings (Column Chart)
- ✓ Business Use: Analyze customer satisfaction and identify improvement areas.

## Real-Life Mini Project: Employee Performance Dashboard

### Project Overview:

A company wants to analyze employee performance data using Power BI. The HR department needs an interactive dashboard that tracks employee work hours, performance ratings, and department-wise analysis.

### Day 85 Tasks

#### Task 1: Understanding Power BI and Its Use Cases

- Research what Power BI is and why businesses use it for analytics.
- Identify the key benefits of Power BI in HR analytics.

#### Task 2: Install Power BI Desktop

- Download and install Power BI Desktop from the official website.
- Verify installation by opening Power BI.

### Task 3: Understand Power BI Components

- Explain the role of Power BI Desktop, Service, Mobile, and Gateway.
- Identify how HR teams can use each component.

### Task 4: Explore Power BI Interface

- Open Power BI and familiarize yourself with the Report View, Data View, and Model View.
- Understand the functions of the Fields Pane, Visualizations Pane, and Filters Pane.

### Task 5: Import an Excel Dataset

- Download a sample employee performance dataset (Excel file).
- Import the dataset into Power BI.

### Task 6: Import a CSV Dataset

- Download and import a CSV file containing additional employee details.
- Verify that the data is correctly loaded.

### Task 7: Clean and Transform Data Using Power Query

- Remove duplicate records and correct formatting issues.
- Ensure column names are meaningful and structured properly.

### Task 8: Create a Data Model

- Establish relationships between the Employee Details and Performance Data tables.
- Ensure one-to-many relationships are properly set.

### Task 9: Create a Summary Table for Employee Performance

- Use DAX (Data Analysis Expressions) to calculate average performance scores.
- Display the top 5 performing employees using a table visualization.

### Task 10: Build a KPI Card for Employee Work Hours

- Create a KPI Card to display total working hours for all employees.
- Set a comparison metric to indicate underperformance.

### Task 11: Create a Department-Wise Performance Chart

- Use a bar chart to display average performance scores by department.
- Apply a color gradient to highlight high and low scores.

### Task 12: Add Filters and Slicers for Interactive Analysis

- Add dropdown filters for selecting departments and job roles.
- Implement a date range slicer for filtering performance data.

### Task 13: Save, Publish, and Share the Report

- Save the Power BI report as a .pbix file.
- Publish the report to Power BI Service.
- Generate a shareable link for the HR team to access the dashboard.

### **Expected Outcome:**

After completing these 13 tasks, you will have a fully functional Employee Performance Dashboard in Power BI. The HR team will be able to:

- ✓ Monitor employee work hours and productivity.

- ✓ Identify top-performing employees.
- ✓ Analyze department-wise performance trends.
- ✓ Make data-driven HR decisions.

## Mini Project 1: Hospital Patient Analysis Dashboard

### Project Requirement:

A hospital wants to analyze patient admission trends, department-wise occupancy rates, and average treatment duration. The management team needs a Power BI dashboard that helps in:

- ✓ Tracking the number of admitted patients per department.
- ✓ Identifying peak admission hours and days.
- ✓ Analyzing average treatment duration per patient.
- ✓ Understanding regional patient distribution.

## Mini Project 2: E-Commerce Order Performance Dashboard

### Project Requirement:

An e-commerce company wants to track order performance, revenue trends, and delivery status using Power BI. The dashboard should provide insights into:

- ✓ Total sales and revenue over different time periods.
- ✓ Top-selling product categories and brands.
- ✓ Order fulfillment rates (delivered, pending, canceled).
- ✓ Region-wise customer orders and sales trends.

# Day 86

## Connecting to Data Sources in Power BI - Step-by-Step Guide

Power BI allows users to connect, transform, and visualize data from multiple sources. This guide will help you understand supported data sources, connection methods, and data refresh techniques with step-by-step examples.

### 1. Supported Data Sources in Power BI

Power BI supports multiple data sources, including:

- ✓ Excel – Import spreadsheets (.xlsx, .xlsm) for analysis.
- ✓ CSV – Load structured data from CSV files.
- ✓ SQL Server – Connect to on-premise or cloud SQL databases.
- ✓ Web – Extract data from websites or online APIs.
- ✓ SharePoint – Retrieve data from SharePoint lists.
- ✓ Azure & Cloud Services – Use Azure SQL, Google BigQuery, Amazon Redshift, etc.
- ✓ Other Databases – MySQL, PostgreSQL, Oracle, IBM DB2, etc.

### 2. Connecting Power BI to Different Data Sources

We will cover how to connect Power BI to Excel, CSV, SQL Server, and Web data step by step.

#### Example 1: Connecting Power BI to Excel

##### Step 1: Open Power BI and Select Excel as Data Source

1. Open Power BI Desktop.

2. Click on Home → Get Data → Excel.
3. Browse and select your Excel file (e.g., sales\_data.xlsx).
4. Click Open.

### **Step 2: Select the Worksheet/Table**

5. The Navigator window will open.
6. Select the sheet/table you want to import (e.g., Sales\_Report).
7. Click Load to import the data.

### **Step 3: View the Imported Data**

8. The dataset will appear in Fields Pane.
9. You can now build visualizations using this data.

## **Example 2: Connecting Power BI to a CSV File**

### **Step 1: Select CSV as Data Source**

1. Open Power BI Desktop.
2. Click Home → Get Data → Text/CSV.
3. Select your CSV file (e.g., customer\_data.csv).
4. Click Open.

### **Step 2: Preview and Load Data**

5. Power BI will display a preview of the CSV file.
6. Click Transform Data to clean data (optional).
7. Click Load to import.

### **Example 3: Connecting Power BI to SQL Server**

#### **Step 1: Open Power BI and Select SQL Server**

1. Open Power BI Desktop.
2. Click Home → Get Data → SQL Server.

#### **Step 2: Enter Server Details**

3. In the SQL Server database window:
  - Enter Server Name (e.g., localhost or 192.168.1.10).
  - Enter Database Name (optional).
4. Choose the connection mode: Import or DirectQuery.

#### **Step 3: Select Authentication Type**

5. Choose how you want to connect:
  - Windows Authentication (if using local login).
  - SQL Server Authentication (enter username & password).

#### **Step 4: Select Tables and Load Data**

6. Select the tables you need (e.g., Orders, Customers).
7. Click Load or Transform Data (if cleaning is needed).

### **Example 4: Connecting Power BI to Web Data (API & Web Scraping)**

#### **Step 1: Open Power BI and Select Web**

1. Click Home → Get Data → Web.
2. Enter the URL of the web data source (e.g., an API or website table).
3. Click OK.

#### **Step 2: Authenticate if Needed**

4. If the website/API requires login, enter your credentials.

5. Click Connect.

### **Step 3: Load or Transform Data**

6. Select the required table/data from the preview window.

7. Click Load to import.

## **3. Using DirectQuery vs Import Mode**

Power BI provides two data connectivity modes:

### **Import Mode**

- Loads a copy of the data into Power BI.
- Faster for large datasets.
- Requires manual or scheduled refresh.

### **DirectQuery Mode**

- Live connection to the data source.
- No data is stored in Power BI.
- Always shows the latest data.
- Requires a stable internet connection.

### **When to Use Import Mode?**

- ✓ Large datasets that need fast performance.
- ✓ When frequent data refresh is manageable.

### **When to Use DirectQuery?**

- ✓ Real-time dashboards where latest data is needed.
- ✓ Cloud-based or SQL Server databases with live updates.

## 4. Refreshing Datasets & Scheduling Automatic Refresh

Power BI allows users to refresh data manually or schedule automatic refreshes.

### Manually Refresh Data

1. Click Home → Refresh to update datasets.

### Schedule Automatic Refresh in Power BI Service

1. Publish the Power BI report to Power BI Service.
2. Go to Power BI Service → Datasets.
3. Click Scheduled Refresh.
4. Set Refresh Frequency (Daily, Hourly, etc.).
5. Click Apply.

## Summary

- ◆ Power BI supports multiple data sources (Excel, SQL, Web, APIs, SharePoint).
- ◆ You can connect Power BI to Excel, CSV, SQL Server, and Web data step by step.
- ◆ Import Mode vs DirectQuery determines how data is loaded.
- ◆ You can manually refresh or schedule automatic refresh in Power BI Service.

## Mini Project 1: Sales Performance Dashboard (Excel & SQL Server Data Integration)

### Project Overview

A retail company wants to analyze sales performance by integrating data from:

- ❖ Excel – Contains monthly sales targets for each region.

✓ **SQL Server** – Stores real-time sales transactions.

The goal is to build a Power BI dashboard that provides insights into:

- ✓ Total sales vs target per region.
- ✓ Top-performing products.
- ✓ Revenue trends over time.

## Step-by-Step Implementation

### Step 1: Connect Power BI to an Excel File (Sales Targets Data)

1. Open Power BI Desktop.
2. Click Home → Get Data → Excel.
3. Select the file Sales\_Targets.xlsx and click Open.
4. Choose the worksheet containing sales targets and click Load.

### Step 2: Connect Power BI to SQL Server (Sales Transactions Data)

1. Click Home → Get Data → SQL Server.
2. Enter your SQL Server Name (e.g., localhost or 192.168.1.10).
3. Choose Database Name (RetailSalesDB).
4. Select Import Mode (for better performance).
5. Click Load to import the Sales\_Transactions table.

### Step 3: Clean and Transform Data Using Power Query

1. Open Power Query Editor (Transform Data).

2. Rename columns for clarity (e.g., Region\_Name instead of Region).
3. Remove null values and duplicate records.
4. Click Close & Apply to save changes.

#### **Step 4: Create Data Relationships**

1. Go to Model View.
2. Drag and link Region\_ID in both tables (Sales\_Targets & Sales\_Transactions).
3. Ensure one-to-many relationship is set.

#### **Step 5: Create Key Metrics Using DAX**

##### **Calculate Total Sales**

TotalSales = SUM(Sales\_Transactions[Amount])

##### **Calculate Target Achievement %**

TargetAchieved = DIVIDE([TotalSales], SUM(Sales\_Targets[TargetAmount])) \* 100

#### **Step 6: Build Visualizations**

KPI Card – Shows total sales.

Bar Chart – Sales performance per region.

Line Chart – Monthly sales trends.

#### **Step 7: Refreshing and Scheduling Automatic Updates**

1. Manually Refresh Data – Click Refresh in Power BI Desktop.

## 2. Schedule Auto-Refresh

- Publish to Power BI Service.
- Go to Datasets → Scheduled Refresh.
- Set refresh frequency (e.g., daily, hourly).

### Expected Outcome

- ✓ A Sales Performance Dashboard displaying real-time sales data.
- ✓ Automated updates from SQL Server & Excel.
- ✓ Interactive reports for regional managers.

## Mini Project 2: Web Scraping & API Data Integration for Stock Market Analysis

### Project Overview

A financial analyst wants to track live stock market prices and historical trends using Power BI by integrating:

- ✓ Web Scraping – Extracting stock data from a website.
- ✓ API – Connecting to a live stock market API.

The dashboard will provide:

- ✓ Real-time stock prices.
- ✓ Stock performance over the past 6 months.
- ✓ High & low price variations.

## Step-by-Step Implementation

### Step 1: Connect Power BI to a Web Data Source

1. Open Power BI Desktop.
2. Click Home → Get Data → Web.
3. Enter the URL of a stock market website (e.g., <https://finance.yahoo.com/>).
4. Click OK, then Select Table containing stock prices.
5. Click Load to import data.

### Step 2: Connect Power BI to a Stock Market API

1. Click Home → Get Data → Web.
2. Enter the API URL (e.g., [https://api.example.com/stock\\_prices](https://api.example.com/stock_prices)).
3. If authentication is required, enter API Key.
4. Click Connect → Transform Data.
5. Remove unnecessary columns and format the data.
6. Click Close & Apply.

### Step 3: Using DirectQuery vs Import Mode

1. For real-time updates, choose DirectQuery Mode.
2. For performance optimization, choose Import Mode.

#### **Step 4: Creating DAX Measures**

##### **Calculate Daily Stock Price Change**

DailyChange = [ClosingPrice] - [OpeningPrice]

##### **Calculate % Change Over Time**

PercentageChange = DIVIDE([DailyChange], [OpeningPrice]) \* 100

#### **Step 5: Build Interactive Visuals**

Table – Displays real-time stock prices.

Line Chart – Stock price trends over 6 months.

KPI Card – Shows highest & lowest stock values.

#### **Step 6: Schedule Automatic Data Refresh**

1. Publish to Power BI Service.
2. Enable API refresh schedules.
3. Set update frequency to every 15 minutes.

#### **Expected Outcome**

- ✓ A Live Stock Market Dashboard.
- ✓ Automatically updated stock prices using APIs & web data.
- ✓ Insights into stock price trends & volatility.

#### **Summary**

- Project 1: Integrated Excel & SQL Server data for sales analysis.

- Project 2: Used Web Scraping & APIs for stock market tracking.
- Both projects demonstrated DirectQuery, Import Mode, and scheduled refresh.

## Mini Project: Customer Feedback Analysis Dashboard

### Project Overview

A company wants to analyze customer feedback collected from different sources:

- ✓ Excel – Customer survey responses.
- ✓ SQL Server – Support ticket data.
- ✓ Web API – Online reviews from a third-party website.

The goal is to build a Power BI dashboard that provides insights into:

- ✓ Customer satisfaction trends.
- ✓ Common issues reported in support tickets.
- ✓ Overall sentiment from online reviews.

### Day 86 Tasks

1. Install Power BI Desktop and ensure it is properly set up.
2. Connect Power BI to an Excel file containing customer survey responses.
3. Connect Power BI to SQL Server to retrieve customer support ticket data.
4. Fetch real-time online customer reviews using a Web API.
5. Compare DirectQuery vs Import Mode for SQL Server and Web API connections.
6. Transform and clean the data by removing duplicates and handling null values.

7. Merge datasets using Power Query to create a unified customer feedback table.
8. Create calculated columns and measures using DAX (e.g., average rating, issue frequency).
9. Build visualizations such as bar charts, sentiment analysis pie charts, and KPI cards.
10. Implement dynamic filters and slicers for better interactivity.
11. Set up automatic data refresh for real-time updates.
12. Publish the report to Power BI Service for online access.
13. Schedule daily or hourly data refresh in Power BI Service.

## **Mini Project 1: Employee Performance Dashboard**

### **Project Requirement:**

A company wants to analyze employee performance by integrating data from multiple sources:

- ✓ Excel – Employee work hours and project details.
- ✓ SQL Server – Performance review scores and attendance records.
- ✓ SharePoint – Training completion status for employees.

The goal is to build a Power BI dashboard that provides insights into:

- ✓ Employee productivity and efficiency.
- ✓ Attendance trends and absenteeism rates.
- ✓ Training progress and skill development.

## Mini Project 2: E-commerce Order Tracking Dashboard

### Project Requirement:

An online store wants to track customer orders and delivery status by integrating data from:

- ✓ CSV Files – Daily order transactions from multiple vendors.
- ✓ SQL Server – Customer order history and payment details.
- ✓ Web API – Real-time shipment tracking updates.

The goal is to create a Power BI dashboard that provides insights into:

- ✓ Total sales and revenue trends.
- ✓ Order processing times and delays.
- ✓ Shipment tracking and delivery success rates.

## Day 87

### Data Cleaning & Transformation in Power BI (Power Query)

#### Introduction to Power Query Editor

Power Query Editor is a data transformation tool in Power BI used to clean, reshape, and transform data before analysis. It allows users to:

- ✓ Remove errors, missing values, and duplicates.
- ✓ Merge, split, and create new columns.
- ✓ Reshape datasets using pivoting and unpivoting.
- ✓ Change data types for better consistency.

## How to Open Power Query Editor?

1. Open Power BI Desktop.
2. Click Home → Transform Data (This opens the Power Query Editor).
3. The Query Editor window appears, where you can see:
  - Query Pane (Left Side) – Lists imported datasets.
  - Data Preview (Center) – Displays the dataset.
  - Applied Steps (Right Side) – Shows the transformations applied.

## Handling Missing Values, Duplicates, and Errors

### Removing Missing Values (Nulls)

When datasets contain blank or null values, they can affect analysis.

#### Steps to Remove Null Values

1. Select the column containing null values.
2. Click Transform → Replace Values.
3. In the Replace Values window:
  - Enter null in the Value to Find box.
  - Enter a default value (e.g., "N/A" or 0).
4. Click OK to apply changes.

**Alternative:** Click Remove Rows → Remove Blank Rows to delete them.

### Removing Duplicate Records

Duplicate records may lead to incorrect calculations in reports.

### **Steps to Remove Duplicates**

1. Select the columns that define a unique record.
2. Click Remove Duplicates in the Home tab.
3. Power Query keeps only the first occurrence and removes others.

### **Handling Errors in Data**

Errors may occur due to incorrect data types or missing values.

### **Steps to Fix Errors**

1. Click the Error Cell to check the error message.
2. Use Replace Errors (under Transform) to replace them with a default value.
3. If needed, change the column's data type using Data Type dropdown.

### **Splitting & Merging Columns**

#### **Splitting Columns (e.g., separating Full Name into First Name and Last Name)**

**Scenario:** A dataset contains Full Name like "John Doe", and we need to split it into "John" and "Doe".

### **Steps to Split a Column**

1. Select the Full Name column.
2. Click Transform → Split Column → By Delimiter.
3. Choose Space as the delimiter and click OK.
4. Power BI creates two new columns: "Full Name.1" and "Full Name.2".
5. Rename them as First Name and Last Name.

## Merging Columns (e.g., combining City and State into Location)

**Scenario:** We have separate City and State columns and want to merge them.

### Steps to Merge Columns

1. Select the City and State columns.
2. Click Transform → Merge Columns.
3. Choose a separator (e.g., ", " for "New York, NY").
4. Click OK, and rename the new column as "Location".

## Creating Custom Columns & Changing Data Types

### Creating a Custom Column (Using a Formula)

**Scenario:** We need to calculate the Total Price by multiplying Quantity by Unit Price.

### Steps to Create a Custom Column

1. Click Add Column → Custom Column.
2. Enter a formula:  
 $= [Quantity] * [Unit Price]$
3. Click OK, and the new column "Total Price" is added.

## Changing Data Types

**Scenario:** The "Order Date" column is in text format, and we need it in Date Format.

## Steps to Change Data Type

1. Select the Order Date column.
2. Click the Data Type dropdown in the top menu.
3. Choose Date.
4. Click Close & Apply to save changes.

## Unpivoting & Reshaping Data

### What is Unpivoting Data?

**Scenario:** We have monthly sales in separate columns:

Product	Jan Sales	Feb Sales	Mar Sales
Laptop	200	250	300

We need to convert it into a row-based format:

Product	Month	Sales
Laptop	Jan	200
Laptop	Feb	250
Laptop	Mar	300

### Steps to Unpivot Data

1. Select the Jan, Feb, Mar Sales columns.
2. Click Transform → Unpivot Columns.
3. Power BI converts them into Month and Sales columns.
4. Rename the new column names for clarity.

## Full Step-by-Step Example: Data Cleaning in Power BI

### Problem Statement:

A company provides an Excel file containing sales data with errors.

We need to clean and prepare the dataset before analysis.

### Raw Data Example:

Order ID	Full Name	City	State	Quantity	Unit Price	Sales	Order Date
1001	John Doe	New York	NY	2	50	Error	2024-01-10
1002	Jane Smith	Los Angeles	CA	5	30	150	2024-02-15
1003	NULL	Chicago	IL	3	40	120	2024-03-12
1003	NULL	Chicago	IL	3	40	120	2024-03-12

### Steps to Clean & Transform Data

1. Remove Duplicate Rows (Order ID 1003 appears twice).
2. Handle Missing Values (Replace NULL in Full Name with "Unknown").
3. Fix Errors in Sales Column (Replace "Error" with Quantity \* Unit Price).
4. Split Full Name into "First Name" and "Last Name".
5. Merge City & State into "Location" ("New York, NY").
6. Change Data Type for "Order Date" to Date Format.
7. Create a New Column "Total Revenue" = Quantity \* Unit Price.
8. Unpivot Sales Data to convert multiple month columns into row format.
9. Close & Apply the transformed data for Power BI reports.

## Summary

- ✓ Power Query is used for cleaning and transforming data before visualization.
- ✓ We covered:
  - Removing null values, duplicates, and errors.
  - Splitting & merging columns.
  - Creating custom columns and changing data types.
  - Unpivoting and reshaping data.
- ✓ These steps ensure clean and structured data for accurate analysis in Power BI.

## Real-Life Mini Project 1: Cleaning & Transforming Sales Data in Power Query

### Project Requirement:

A retail company collects monthly sales data from different stores. However, the dataset has missing values, duplicate records, incorrect data types, and is not structured properly. You need to clean, transform, and reshape the data using Power Query in Power BI.

### Step-by-Step Implementation:

#### Step 1: Import Dataset into Power BI

1. Open Power BI Desktop.
2. Click Home → Get Data → Excel.
3. Select the sales dataset and Load Data.
4. Click Transform Data to open Power Query Editor.

### Sample Raw Data (Before Cleaning)

Order ID	Full Name	Store Name	City	State	Quantity	Unit Price	Sales	Order Date
1001	John Doe	Store A	New York	NY	2	50	Error	2024-01-10
1002	Jane Smith	Store B	Los Angeles	CA	5	30	150	2024-02-15
1003	NULL	Store C	Chicago	IL	3	40	120	2024-03-12
1003	NULL	Store C	Chicago	IL	3	40	120	2024-03-12

### Step 2: Remove Duplicates

**Problem:** Order ID 1003 appears twice.

#### Solution:

1. Select the Order ID column.
2. Click Remove Duplicates in the Home tab.
3. Power Query removes duplicate rows.

### Step 3: Handle Missing Values

**Problem:** Full Name is missing (NULL) for some records.

#### Solution:

1. Select the Full Name column.
2. Click Transform → Replace Values.
3. Enter null in Value to Find and "Unknown" in Replace With.
4. Click OK.

#### **Step 4: Fix Data Errors (Sales Column)**

**Problem:** The Sales column contains "Error".

**Solution:**

1. Select the Sales column.
2. Click Replace Errors → Enter Quantity \* Unit Price.
3. Click OK.

#### **Step 5: Split Full Name into First & Last Name**

**Problem:** Full Name should be separated into First Name and Last Name.

**Solution:**

1. Select the Full Name column.
2. Click Transform → Split Column → By Delimiter.
3. Choose Space as the delimiter.
4. Rename the new columns: "First Name" and "Last Name".

#### **Step 6: Merge City & State into Location**

**Problem:** Need to combine City and State into "Location".

**Solution:**

1. Select the City and State columns.
2. Click Transform → Merge Columns.
3. Choose ", " as the separator.
4. Rename the new column as "Location".

### Step 7: Change Data Type for Order Date

**Problem:** The "Order Date" column is in text format.

**Solution:**

1. Select Order Date column.
2. Click the Data Type dropdown.
3. Choose Date.

### Step 8: Create a New Column - Total Revenue

**Problem:** Need to calculate **Total Revenue** = Quantity \* Unit Price.

**Solution:**

1. Click Add Column → Custom Column.
2. Enter formula:  
 $= [Quantity] * [Unit Price]$
3. Click OK.

### Step 9: Unpivot Monthly Sales Data

**Problem:** Monthly sales data is in column format, but it should be in row format.

**Solution:**

1. Select the monthly sales columns (Jan, Feb, Mar, etc.).
2. Click Transform → Unpivot Columns.
3. Rename the new column names:
  - Attribute → Month
  - Value → Sales

### Step 10: Apply Changes & Load Data

1. Click Close & Apply in Power Query.
2. The cleaned and transformed dataset is now available in Power BI for visualization.

#### Final Dataset (After Cleaning & Transformation):

Order ID	First Name	Last Name	Store Name	Location	Quantity	Unit Price	Total Revenue	Month	Sales
1001	John	Doe	Store A	New York, NY	2	50	100	Jan	200
1002	Jane	Smith	Store B	Los Angeles, CA	5	30	150	Feb	250
1003	Unknown	-	Store C	Chicago, IL	3	40	120	Mar	300

### Real-Life Mini Project 2: Cleaning Employee Data using Power Query

#### Project Requirement:

An HR department collects employee records but faces data quality issues.

You need to clean, transform, and restructure the dataset for better reporting.

#### Raw Dataset Issues:

- ✓ Missing values in Department & Salary.
- ✓ Duplicates in Employee ID.
- ✓ Inconsistent formatting in Joining Date.
- ✓ Full Name needs splitting into First Name & Last Name.

- ✓ Department Names need standardization.

## **Steps to Clean & Transform Employee Data:**

### **Step 1: Import Employee Dataset**

- ✓ Go to Home → Get Data → Excel/CSV → Load the file.
- ✓ Click Transform Data to open Power Query Editor.

### **Step 2: Remove Duplicate Employees**

- ✓ Select the Employee ID column → Click Remove Duplicates.

### **Step 3: Handle Missing Values**

- ✓ Replace NULL values in Department with "Unknown".
- ✓ Replace NULL values in Salary with the department's average salary:  
= if [Salary] = null then List.Average(#"Previous Step"[Salary]) else [Salary]

### **Step 4: Standardize Department Names**

- ✓ Use Replace Values to fix inconsistent department names:
  - "HR Dept" → "Human Resources"
  - "IT-Dept" → "IT"

### **Step 5: Split Full Name into First & Last Name**

- ✓ Transform → Split Column → By Space.
- ✓ Rename new columns "First Name" and "Last Name".

### **Step 6: Fix Date Format for Joining Date**

- ✓ Select Joining Date → Change Data Type to Date.

### **Step 7: Create a New Column - Annual Salary**

- ✓ Add a Custom Column:

= [Salary] \* 12

### **Step 8: Apply Changes & Load Data**

- ✓ Click Close & Apply.
- ✓ Use the cleaned data in Power BI for HR analytics.

### **Summary**

- ✓ Power Query simplifies data cleaning & transformation.
- ✓ Key actions: Removing duplicates, handling missing values, fixing errors, splitting/merging columns, unpivoting data.
- ✓ Now, the data is ready for visualization in Power BI.

### **Real-Life Mini Project: Customer Purchase Data Cleaning & Transformation**

#### **Project Requirement:**

A retail company has collected customer purchase data, but it contains inconsistencies and errors. The dataset needs cleaning, transformation, and restructuring for better analysis in Power BI.

## Day 87 Tasks

1. Import the dataset into Power BI from an Excel or CSV file.
2. Open Power Query Editor and explore the dataset.
3. Remove duplicate records based on Customer ID and Order ID.
4. Handle missing values in the Customer Name, Product, and Purchase Amount columns.
5. Fix data errors (e.g., incorrect purchase amounts labeled as "Error").
6. Split Full Name into First Name & Last Name for better organization.
7. Merge City & State columns to create a new "Location" column.
8. Convert data types (e.g., ensure Purchase Amount is in decimal format, Order Date is in date format).
9. Create a new calculated column for "Total Purchase Value" = Purchase Quantity \* Unit Price.
10. Standardize product category names (e.g., "Mobiles" vs "Mobile Phones").
11. Unpivot sales data (if monthly sales are stored in separate columns) to make it more structured.
12. Remove unwanted columns (e.g., columns that do not contribute to analysis).
13. Apply changes & load cleaned data back into Power BI for reporting.

### Final Outcome:

After completing these tasks, the dataset will be clean, structured, and ready for further analysis and visualization in Power BI.

## Mini Project 1: Sales Data Cleanup & Transformation

### Scenario:

A retail company has sales data from multiple stores, but it contains inconsistencies, missing values, and format issues. The data needs to be cleaned and transformed for accurate reporting.

### Key Requirements:

- Remove duplicate sales entries based on Invoice Number.
- Handle missing values in Customer Name, Product Name, and Sales Amount columns.
- Fix incorrect date formats in the transaction date column.
- Merge Store Location & Region columns into a single field.
- Split the Full Name column into First Name and Last Name.
- Standardize product categories (e.g., "TVs" vs. "Televisions").
- Convert Sales Amount and Discounts to proper numerical types.
- Unpivot monthly sales data into a structured format.
- Apply necessary transformations and load the cleaned dataset into Power BI.

## Mini Project 2: Employee Data Cleanup & Reshaping

### Scenario:

An HR department has a dataset containing employee records but needs to clean and reshape it for better analysis in Power BI.

### Key Requirements:

- Remove duplicate employee records based on Employee ID.
- Fill missing values in Department and Salary columns.
- Convert salary details to currency format.
- Split the Full Name column into First Name and Last Name.
- Merge City and State into a single Location column.

- Standardize Job Titles (e.g., "Software Eng." → "Software Engineer").
- Unpivot department-wise headcount data for better analysis.
- Create a new calculated column for Annual Salary = Monthly Salary × 12.
- Load the cleaned dataset into Power BI for HR analytics.

## Day 88

### Data Modeling & Relationships in Power BI

#### What is Data Modeling in Power BI?

Data modeling in Power BI is the process of structuring and relating multiple tables to create meaningful insights. It allows users to define relationships between datasets, ensuring efficient querying, calculations, and reporting.

#### Why is Data Modeling Important?

- Improves performance by reducing data redundancy.
- Helps in creating accurate reports & dashboards.
- Enables better querying and filtering of data.
- Establishes relationships between different tables for better insights.

### Understanding Star Schema & Snowflake Schema

#### 1. Star Schema

- A central fact table (contains measurable data) is connected to multiple dimension tables (categories like customers, products, dates).
- Simpler and faster for reporting.

**Example:** A sales database with

- **Fact Table:** Sales Transactions

- **Dimension Tables:** Customers, Products, Stores, Dates

## 2. Snowflake Schema

- A variation of the Star Schema where dimension tables are further normalized into sub-dimensions.
- Reduces data redundancy but increases query complexity.

### Example:

- Fact Table: Sales Transactions
- Dimension Tables: Products (which is linked to Product Category)

## Creating Relationships Between Tables

Power BI allows users to create relationships between tables based on common fields.

### Steps to Create a Relationship in Power BI

1. Load multiple datasets (Excel, SQL, etc.).
2. Open Model View in Power BI.
3. Drag & drop a field from one table to another to create a relationship.
4. Define the relationship type (One-to-Many, Many-to-One, etc.).
5. Validate and save the relationship.

### Example:

- Table 1: Orders (Order ID, Customer ID, Product ID, Amount)
- Table 2: Customers (Customer ID, Name, Region)
- Relationship: Orders → Customers (One-to-Many on Customer ID)

## Using Primary Key & Foreign Key Concepts

- Primary Key: A unique identifier in a table (e.g., Customer ID in the Customers table).

- Foreign Key: A field in another table that references a Primary Key (e.g., Customer ID in the Orders table).

**Example:**

- Customers Table:
  - Customer ID (Primary Key)
- Orders Table:
  - Customer ID (Foreign Key, linking to Customers table)

## **Handling Many-to-One & One-to-Many Relationships**

### **Relationship Types in Power BI**

1. One-to-One (1:1) → A single row in Table A matches a single row in Table B.
2. One-to-Many (1:M) → A single row in Table A matches multiple rows in Table B.
3. Many-to-One (M:1) → The reverse of One-to-Many.
4. Many-to-Many (M:M) → A complex relationship where multiple rows match multiple rows.

**Example:**

- One-to-Many: A single Customer (Customers Table) can place multiple Orders (Orders Table).
- Many-to-One: Multiple Orders belong to one Customer.

### **How to Set Relationships in Power BI**

1. Go to Model View.
2. Drag a field (e.g., Customer ID) from Orders to Customers.
3. Choose One-to-Many relationship type.
4. Enable "Referential Integrity" if applicable.

## Using Calculated Columns

A calculated column is a new column created using DAX (Data Analysis Expressions).

### Syntax:

ColumnName = TableName[Column1] + TableName[Column2]

### Example: Create a Total Sales Column

Total Sales = Orders[Quantity] \* Orders[Unit Price]

- This formula creates a new column calculating the total price of each order.

## Step-by-Step Real-Life Example

### Scenario: E-Commerce Sales Data Analysis

#### Step 1: Import Data

- Load Orders.csv, Customers.csv, and Products.csv into Power BI.

#### Step 2: Create Relationships

- Orders[Customer ID] → Customers[Customer ID] (One-to-Many)
- Orders[Product ID] → Products[Product ID] (One-to-Many)

#### Step 3: Add a Calculated Column

- Add "Total Revenue" in the Orders table:

Total Revenue = Orders[Quantity] \* Orders[Unit Price]

#### Step 4: Create a Report

- Use **bar charts, tables, and KPIs** to visualize total sales, top customers, and product demand.

## Final Outcome

By applying relationships, calculated columns, and modeling best practices, the E-Commerce Sales Dashboard in Power BI will provide powerful insights into sales performance.

## Mini Project 1: E-Commerce Sales Data Modeling

### Scenario:

A growing e-commerce business wants to analyze its sales, customers, and products using Power BI. However, the data is stored across multiple tables and needs proper data modeling for insights.

### Step 1: Import Data into Power BI

Load the following datasets into Power BI:

1. Orders.csv → Order details with Order ID, Customer ID, Product ID, Order Date, Quantity, Unit Price, Total Sales
2. Customers.csv → Customer ID, Customer Name, Region, Country
3. Products.csv → Product ID, Product Name, Category, Brand, Price
4. Dates.csv → Date, Year, Month, Weekday, Quarter

### Step 2: Apply Star Schema

- Fact Table: Orders (contains numerical sales data)
- Dimension Tables: Customers, Products, Dates

This setup follows the Star Schema because we have a central fact table (Orders) and multiple dimension tables.

### Step 3: Define Relationships

1. Orders[Customer ID] → Customers[Customer ID] (One-to-Many)
2. Orders[Product ID] → Products[Product ID] (One-to-Many)
3. Orders[Order Date] → Dates[Date] (One-to-Many)
- Power BI automatically detects relationships, but we can manually adjust them in Model View.

### Step 4: Using Primary & Foreign Keys

- Customer ID is a Primary Key in Customers and a Foreign Key in Orders.
- Product ID is a Primary Key in Products and a Foreign Key in Orders.
- Date is a Primary Key in Dates and a Foreign Key in Orders.

### Step 5: Creating Calculated Columns

#### 1. Total Revenue Calculation

Total Revenue = Orders[Quantity] \* Orders[Unit Price]

#### 2. Profit Calculation

Profit = Orders[Total Sales] - (Orders[Quantity] \* Products[Price] \* 0.7)

(Assuming cost is 70% of product price)

### Step 6: Creating a Dashboard

1. Add KPIs for Total Revenue, Total Orders, Total Customers.
2. Create a Bar Chart for Sales by Category.
3. Create a Map Visualization for Sales by Country.

### Step 7: Implement Automatic Refresh

- Schedule a refresh in Power BI Service to ensure the latest data updates.

**Final Output:** A fully interactive E-Commerce Sales Dashboard in Power BI!

## Mini Project 2: HR Employee Database & Analysis

### Scenario:

An HR department wants to analyze employee distribution, salaries, and department data using Power BI. The data is stored in separate tables and needs proper data modeling.

### Step 1: Import Data into Power BI

Load the following datasets:

1. Employees.csv → Employee ID, Full Name, Department ID, Salary, Join Date, Gender
2. Departments.csv → Department ID, Department Name, Manager ID
3. Managers.csv → Manager ID, Manager Name
4. Dates.csv → Date, Year, Month, Quarter

### Step 2: Apply Snowflake Schema

Unlike the Star Schema, here we normalize tables further:

- Fact Table: Employees
- Dimension Tables: Departments, Managers, Dates

### Why Snowflake Schema?

The Managers table is a sub-dimension of Departments, reducing redundancy.

### Step 3: Define Relationships

1. Employees[Department ID] → Departments[Department ID] (One-to-Many)
2. Departments[Manager ID] → Managers[Manager ID] (One-to-One)
3. Employees[Join Date] → Dates[Date] (One-to-Many)
- Manager ID ensures that each department has one manager.

### Step 4: Using Primary & Foreign Keys

- Employee ID is a Primary Key in Employees.
- Department ID is a Primary Key in Departments and a Foreign Key in Employees.
- Manager ID is a Primary Key in Managers and a Foreign Key in Departments.

### Step 5: Creating Calculated Columns

#### 1. Annual Salary Calculation

Annual Salary = Employees[Salary] \* 12

#### 2. Years of Service Calculation

Years of Service = YEAR(TODAY()) - YEAR(Employees[Join Date])

### Step 6: Creating a Dashboard

1. KPI Cards for Total Employees, Avg Salary, Total Departments.
2. Bar Chart for Employee Count by Department.
3. Pie Chart for Gender Distribution.
4. Line Chart for Hiring Trend Over the Years.

### Step 7: Implement Automatic Refresh

- Schedule Power BI Refresh for updated HR analytics.

**Final Output:** A powerful HR Analytics Dashboard in Power BI! 🎉

## Real-Life Mini Project: Retail Store Sales Analysis

### Objective:

Create a Power BI dashboard for a retail store to analyze sales, customers, and product performance using proper data modeling and relationships.

### Day 88 Tasks

#### Task 1: Import Sales Data

Load sales data from an Excel or CSV file into Power BI.

#### Task 2: Import Customer Data

Load customer details, including Customer ID, Name, Age, Gender, and Location.

#### Task 3: Import Product Data

Load product details, including Product ID, Name, Category, Price, and Brand.

#### Task 4: Import Date Table

Create or load a Date Table with Date, Year, Month, Quarter, and Weekday fields.

### Task 5: Design a Star Schema

Structure the dataset using a Star Schema with:

- Fact Table: Sales
- Dimension Tables: Customers, Products, Dates

### Task 6: Establish Relationships

Define relationships between tables based on Primary Key & Foreign Key constraints.

### Task 7: Define Many-to-One Relationships

Set up relationships such as:

- Sales[Customer ID] → Customers[Customer ID]
- Sales[Product ID] → Products[Product ID]
- Sales[Date] → Dates[Date]

### Task 8: Create a Calculated Column for Total Sales

Use DAX to calculate total sales as Quantity \* Price.

### Task 9: Create a Profit Calculation Column

Define a Profit Calculation assuming a fixed cost percentage.

### Task 10: Build a Sales by Category Report

Create a Bar Chart showing total sales by product category.

### Task 11: Build a Customer Demographics Report

Create a Pie Chart to visualize the gender distribution of customers.

### Task 12: Create a Sales Trend Analysis

Develop a Line Chart to show sales trends over time.

### Task 13: Schedule Data Refresh

Enable automatic refresh for real-time data updates in Power BI Service.

## **1. Real-Life Mini Project: Hospital Patient Management System**

### **Project Objective:**

Develop a Power BI dashboard to analyze hospital patient records, including appointments, doctors, treatments, and billing details, using data modeling techniques.

### **Key Requirements:**

- Import and structure patient, doctor, appointment, and billing data.
- Use a Star Schema with a fact table for appointments and dimension tables for patients, doctors, and treatments.
- Define Primary Key & Foreign Key relationships between tables.
- Implement one-to-many relationships (e.g., one doctor treating multiple patients).
- Create calculated columns for total revenue generated per patient.
- Build a dashboard showing appointment trends, patient demographics, and hospital revenue insights.

## 2. Real-Life Mini Project: E-Commerce Order Processing Analysis

### Project Objective:

Design a Power BI report to analyze order processing, customer demographics, and product performance for an e-commerce company.

### Key Requirements:

- Import order, customer, product, and shipping data.
- Use a Star Schema where Orders is the fact table, and Customers, Products, and Dates are dimension tables.
- Establish relationships between tables using Customer ID, Product ID, and Order Date.
- Implement many-to-one relationships (e.g., one customer placing multiple orders).
- Create calculated columns for total order value, shipping cost, and profit margin.
- Develop visualizations for sales trends, best-selling products, and order fulfillment times.

## Day 89

### Introduction to DAX (Data Analysis Expressions) in Power BI

#### 1. What is DAX? Why use it?

##### Definition:

DAX (Data Analysis Expressions) is a formula language used in Power BI, Power Pivot, and SQL Server Analysis Services (SSAS) to perform advanced calculations and create custom aggregations in reports.

##### Why use DAX?

- ◆ Helps perform complex calculations and business logic.

- ◆ Enables advanced data modeling and filtering.
- ◆ Allows the creation of custom measures, calculated columns, and tables.
- ◆ Supports aggregation functions like SUM, AVERAGE, COUNT, etc.

## 2. Understanding Measures vs. Calculated Columns

Feature	Measures	Calculated Columns
Definition	Computed at the time of report generation	Computed at the row level when data is loaded
Storage	Not stored in the dataset, calculated dynamically	Stored in the dataset
Performance	Faster, as calculations happen on the fly	Slower, as each row stores a computed value
Best for	Aggregations (SUM, COUNT, AVERAGE, etc.)	Row-based computations (New columns, IF conditions, etc.)
Example	Total Sales = SUM(Sales[Amount])	Profit = Sales[Revenue] - Sales[Cost]

## 3. Common DAX Functions with Examples

### SUM() - Calculate Total Sales

#### Syntax:

`SUM(<ColumnName>)`

#### Example:

Calculate the total sales amount from the Sales table.

`Total Sales = SUM(Sales[Amount])`

#### Real-Life Scenario:

A retail manager wants to see the total revenue generated by the store.

## **AVERAGE() - Calculate Average Sales**

### **Syntax:**

AVERAGE(<ColumnName>)

### **Example:**

Find the average order value.

Avg Order Value = AVERAGE(Sales[Amount])

### **Real-Life Scenario:**

An e-commerce company wants to track the average order value per transaction.

## **COUNT() - Count Number of Transactions**

### **Syntax:**

COUNT(<ColumnName>)

### **Example:**

Count the number of orders.

Total Orders = COUNT(Sales[Order ID])

### **Real-Life Scenario:**

A store manager wants to count the total number of orders placed daily.

## **DISTINCTCOUNT() - Count Unique Customers**

### **Syntax:**

DISTINCTCOUNT(<ColumnName>)

### **Example:**

Find the number of unique customers.

Unique Customers = DISTINCTCOUNT(Sales[Customer ID])

### **Real-Life Scenario:**

A business analyst wants to know how many different customers made a purchase.

### **IF() - Conditional Calculation**

#### **Syntax:**

IF(<Condition>, <True Result>, <False Result>)

#### **Example:**

Create a column to **classify sales performance**.

Sales Performance = IF(Sales[Amount] > 1000, "High", "Low")

### **Real-Life Scenario:**

A manager wants to label orders as "High" sales or "Low" sales based on order amount.

### **SWITCH() - Multiple Conditions**

#### **Syntax:**

SWITCH(Expression, Value1, Result1, Value2, Result2, ..., ElseResult)

#### **Example:**

Classify customers based on age groups.

Customer Age Group = SWITCH(  
TRUE(),  
Customers[Age] < 18, "Teen",  
Customers[Age] >= 18 && Customers[Age] <= 35, "Young Adult",  
Customers[Age] > 35, "Adult"  
)

**Real-Life Scenario:**

A marketing team wants to segment customers into different age groups.

**RANKX() - Rank Products by Sales**

**Syntax:**

`RANKX(<Table>, <Expression>, <Value>, <Order>)`

**Example:**

Rank products by total sales.

Product Rank = `RANKX(ALL(Sales[Product Name]), SUM(Sales[Amount]), , DESC)`

**Real-Life Scenario:**

An inventory manager wants to find the best-selling product.

**FILTER() - Apply Row-Level Filteringing**

**Syntax:**

`FILTER(<Table>, <Condition>)`

**Example:**

Filter only sales above \$500.

High Sales = `FILTER(Sales, Sales[Amount] > 500)`

**Real-Life Scenario:**

A sales director wants to analyze only high-value transactions.

**CALCULATE() - Modify Context of a Measure**

**Syntax:**

`CALCULATE(<Expression>, <Filter>)`

**Example:**

Calculate total sales for 2024 only.

Sales 2024 = CALCULATE(SUM(Sales[Amount]), Sales[Year] = 2024)

**Real-Life Scenario:**

A finance analyst wants to compare sales revenue across different years.

**ALL() - Remove Filters for Total Calculation**

**Syntax:**

ALL(<Table> | <Column>)

**Example:**

Calculate total sales without any filters.

Total Sales (All Data) = CALCULATE(SUM(Sales[Amount]), ALL(Sales))

**Real-Life Scenario:**

A CEO wants to view total sales across all products, regardless of filters.

**Step-by-Step Implementation in Power BI**

**Step 1: Load Data**

1. Open Power BI Desktop
2. Click Home → Get Data → Excel/CSV/SQL Server
3. Load the dataset and click Transform Data

**Step 2: Open Power BI's DAX Editor**

1. Click on Modeling → New Measure
2. Enter your DAX formula (e.g., Total Sales = SUM(Sales[Amount]))
3. Press Enter

### Step 3: Create a Visualization

1. Click Report View
2. Select Bar Chart, Table, or Card
3. Drag the DAX measure to visualize the data

### Summary of Key DAX Functions

Function	Purpose
SUM()	Adds up values in a column
AVERAGE()	Finds the average of a column
COUNT()	Counts the number of rows
DISTINCTCOUNT()	Counts unique values
IF()	Performs conditional logic
SWITCH()	Evaluates multiple conditions
RANKX()	Ranks items based on a value
FILTER()	Filters a table based on a condition
CALCULATE()	Modifies context of a calculation
ALL()	Ignores filters to calculate over all data

### Final Thoughts

DAX is a powerful tool for performing advanced calculations and aggregations in Power BI. By mastering these functions, you can create custom reports, analyze business performance, and generate meaningful insights!

### Mini Project 1: Sales Performance Dashboard using DAX in Power BI

#### Project Requirements

Create a Sales Performance Dashboard that analyzes monthly sales trends, top-selling products, customer insights, and revenue growth using DAX functions in Power BI. The project will include:

- ✓ Total Sales Calculation using SUM()
- ✓ Average Order Value using AVERAGE()
- ✓ Unique Customer Count using DISTINCTCOUNT()
- ✓ Profit Calculation using IF()
- ✓ Product Ranking using RANKX()
- ✓ Sales Growth % Calculation using CALCULATE()

## Step-by-Step Implementation

### Step 1: Load Data into Power BI

1. Open Power BI Desktop
2. Click Home → Get Data → Excel/CSV/SQL Server
3. Select Sales Data.xlsx and Load Data
4. Click Transform Data to clean if necessary

### Sample Sales Dataset (SalesData Table)

Order ID	Date	Product	Category	Amount	Cost	Customer ID
101	2024-01-05	Laptop	Electronics	1200	800	C001
102	2024-01-07	Phone	Electronics	800	500	C002
103	2024-01-10	Mouse	Accessories	50	20	C003
104	2024-02-02	Monitor	Electronics	300	200	C001

### Step 2: Create DAX Measures

1. Total Sales Calculation using SUM()

**Formula:**

Total Sales = SUM(SalesData[Amount])

**Explanation:**

This measure calculates the total revenue from all sales.

2. Average Order Value using AVERAGE()

**Formula:**

Average Order Value = AVERAGE(SalesData[Amount])

**Explanation:**

This measure finds the average revenue per order.

3. Unique Customer Count using DISTINCTCOUNT()

**Formula:**

Unique Customers = DISTINCTCOUNT(SalesData[Customer ID])

**Explanation:**

This calculates the number of unique customers who made a purchase.

4. Profit Calculation using IF()

**Formula:**

Profit = SUM(SalesData[Amount]) - SUM(SalesData[Cost])

**Explanation:**

Calculates the total profit (Revenue - Cost).

## 5. Rank Top Products using RANKX()

### **Formula:**

Product Rank = RANKX(ALL(SalesData[Product]), SUM(SalesData[Amount]), , DESC)

### **Explanation:**

Ranks products based on total sales revenue.

## 6. Sales Growth % using CALCULATE()

### **Formula:**

Sales Last Month = CALCULATE(SUM(SalesData[Amount]), PREVIOUSMONTH(SalesData[Date]))

Sales Growth % = DIVIDE([Total Sales] - [Sales Last Month], [Sales Last Month], 0)

### **Explanation:**

This calculates the percentage growth in sales compared to the previous month.

## **Step 3: Build Dashboard in Power BI**

1. Add a Card Visual for Total Sales, Average Order Value, and Unique Customers
2. Add a Table Visual for Product Rankings
3. Add a Line Chart to show Monthly Sales Trends
4. Add a Gauge Chart for Sales Growth %

## Summary of Outputs

- ✓ Total Sales → \$2,350
- ✓ Average Order Value → \$587.50
- ✓ Unique Customers → 3
- ✓ Top Product → Laptop
- ✓ Sales Growth % → +10%

This dashboard helps businesses track sales trends, customer activity, and product performance using DAX expressions.

## Mini Project 2: Employee Performance Analytics using DAX

### Project Requirements

Create an Employee Performance Dashboard to analyze employee productivity, attendance, and department-wise efficiency using DAX functions in Power BI. The project will include:

- ✓ Total Hours Worked using SUM()
- ✓ Average Hours per Employee using AVERAGE()
- ✓ Unique Employee Count using DISTINCTCOUNT()
- ✓ Employee Rating Calculation using IF()
- ✓ Ranking Employees by Efficiency using RANKX()
- ✓ Filter Employees with Low Attendance using FILTER()
- ✓ Performance Score Calculation using CALCULATE()

## Step-by-Step Implementation

### Step 1: Load Data into Power BI

1. Open Power BI Desktop
2. Click Home → Get Data → Excel/SQL Server
3. Select EmployeeData.xlsx and Load Data

### Sample Employee Dataset (EmployeeData Table)

Employee ID	Name	Department	Hours Worked	Rating	Absent Days
E001	John	HR	160	4.5	2
E002	Alice	IT	180	4.8	1
E003	Bob	Finance	150	4.2	5
E004	Charlie	IT	200	4.9	0

### Step 2: Create DAX Measures

1. Total Hours Worked using SUM()

#### Formula:

Total Hours = SUM(EmployeeData[Hours Worked])

#### Explanation:

Calculates total working hours of all employees.

2. Average Hours per Employee using AVERAGE()

#### Formula:

Avg Hours per Employee = AVERAGE(EmployeeData[Hours Worked])

#### Explanation:

Finds the average working hours per employee.

### 3. Unique Employee Count using DISTINCTCOUNT()

**Formula:**

Total Employees = DISTINCTCOUNT(EmployeeData[Employee ID])

**Explanation:**

Counts the total number of employees.

### 4. Employee Rating Calculation using IF()

**Formula:**

Performance = IF(EmployeeData[Rating] > 4.5, "Excellent", "Good")

**Explanation:**

Categorizes employees based on performance rating.

### 5. Rank Employees by Efficiency using RANKX()

**Formula:**

Employee Rank = RANKX(ALL(EmployeeData[Name]), SUM(EmployeeData[Hours Worked]), , DESC)

**Explanation:**

Ranks employees by total hours worked.

### 6. Filter Employees with Low Attendance using FILTER()

**Formula:**

Low Attendance = FILTER(EmployeeData, EmployeeData[Absent Days] > 3)

**Explanation:**

Filters employees with more than 3 absences.

7. Performance Score using CALCULATE()

**Formula:**

Performance Score = CALCULATE(SUM(EmployeeData[Hours Worked]) - (EmployeeData[Absent Days] \* 5))

**Explanation:**

Calculates a performance score penalizing absenteeism.

**Step 3: Build Dashboard in Power BI**

1. Add a Card Visual for Total Hours, Avg Hours, and Total Employees
2. Add a Table Visual for Top Employees by Rank
3. Add a Pie Chart for Department-Wise Work Distribution
4. Add a Bar Chart for Employee Performance Ratings

**Summary of Outputs**

- ✓ Total Hours Worked → 690
- ✓ Average Hours per Employee → 172.5
- ✓ Top Employee → Charlie
- ✓ Employees with Low Attendance → Bob

This dashboard helps HR managers track employee performance and attendance using DAX calculations.

## Mini Project: Retail Store Sales Analysis using DAX in Power BI

### Project Overview:

A retail store wants to analyze its sales performance using DAX in Power BI. The analysis will include total sales, average sales, customer count, product rankings, and sales trends using various DAX functions.

### Day 89 Tasks

#### Task 1: Load and Prepare Data

- Import Sales Data into Power BI from an Excel or SQL database.
- Clean and transform the data if needed (handle missing values, duplicates).

#### Task 2: Create a Total Sales Measure using SUM()

- Write a DAX measure to calculate the total revenue from all sales.

#### Task 3: Calculate the Average Sales per Order using AVERAGE()

- Find the average sales amount per order.

#### Task 4: Count Unique Customers using DISTINCTCOUNT()

- Use DAX to count the total number of unique customers.

#### Task 5: Identify the Best-Selling Product using RANKX()

- Rank the top-selling products based on total revenue.

#### Task 6: Categorize Products Based on Sales using IF()

- If a product's total sales > \$10,000, mark it as "High Performer"; otherwise, mark it as "Low Performer".

#### Task 7: Create a Sales Category using SWITCH()

- Categorize sales into "Low", "Medium", and "High" based on revenue thresholds.

#### Task 8: Calculate Monthly Sales Growth using CALCULATE()

- Compare current month's sales vs previous month's sales and calculate growth percentage.

Task 9: Find the Sales Contribution of Each Product using ALL()

- Determine how much each product contributes to the total sales percentage.

Task 10: Filter High-Value Orders using FILTER()

- Identify orders with a total amount greater than \$500.

Task 11: Calculate Sales per Customer using a Calculated Column

- Create a calculated column to find sales per customer (Total Sales / Unique Customers).

Task 12: Identify Top 5 Customers Based on Spending

- Rank customers based on total purchase amount and find the top 5 spenders.

Task 13: Build a Power BI Dashboard with Visualizations

- Display total sales, product rankings, customer insights, and sales trends in a Power BI dashboard.

### **Expected Outcome:**

This project will provide insights into sales performance, customer behavior, and product popularity using DAX functions in Power BI.

## **Mini Project 1: Employee Performance & Salary Analysis using DAX in Power BI**

### **Project Requirements:**

A company wants to analyze employee performance and salary data using Power BI and DAX functions. The project should include:

- Total salary expenses calculation using SUM()
- Average salary per department using AVERAGE()
- Unique employee count using DISTINCTCOUNT()
- Employee ranking based on performance score using RANKX()
- Categorizing employees based on salary bands using IF() and SWITCH()
- Filtering high-performing employees using FILTER()
- Calculating department-wise salary contribution using ALL()

## **Mini Project 2: E-commerce Customer Purchase Behavior Analysis using DAX**

### **Project Requirements:**

An e-commerce company wants to analyze customer purchase behavior using Power BI. The project should include:

- Total revenue calculation using SUM()
- Average order value per customer using AVERAGE()
- Unique customer count using DISTINCTCOUNT()
- Ranking of customers based on total purchases using RANKX()
- Customer segmentation into "High-Value", "Medium-Value", and "Low-Value" using IF() and SWITCH()
- Filtering repeat customers using FILTER()
- Calculating the percentage contribution of each customer using ALL()

# Day 90

## Creating Visualizations in Power BI

### 1. What is Power BI Visualization?

Visualization in Power BI refers to the process of representing data using charts, graphs, and interactive elements. Power BI provides a variety of visualization tools to help users understand trends, comparisons, and patterns easily.

#### Why use Visualizations?

- Makes complex data easier to understand
- Identifies trends, outliers, and correlations
- Enhances decision-making with interactive reports
- Allows users to drill deeper into detailed insights

### 2. Different Charts & Graphs in Power BI

Power BI provides a rich set of visualization tools such as:

Chart Type	Best Used For
Bar Chart	Comparing categories
Column Chart	Visualizing trends over time
Line Chart	Tracking trends & patterns
Pie Chart	Showing proportions
Map Chart	Geographical data
Card	Displaying key metrics
Table & Matrix	Showing detailed tabular data
Scatter Plot	Analyzing correlations

### 3. Step-by-Step Implementation of Visualizations in Power BI

#### Example: Sales Performance Dashboard

Let's create a Sales Performance Dashboard using different charts and Power BI features.

#### Step 1: Importing Data

1. Open Power BI Desktop
2. Click on Home → Get Data → Excel
3. Select the sales\_data.xlsx file and load it

#### Example Dataset Columns:

- Order ID (Unique ID)
- Product (Item Name)
- Category (Product Category)
- Sales (Total Sales Amount)
- Profit (Profit Earned)
- Region (Location of Sale)
- Date (Transaction Date)

#### Step 2: Creating Bar Charts

A bar chart is useful for comparing total sales across different product categories.

1. Click on Bar Chart from the Visualizations Pane
2. Drag Category to the X-axis
3. Drag Sales to the Y-axis
4. Format the chart (colors, labels, title)

#### Insights from Bar Chart:

- Which category has the highest sales?
- How do categories compare against each other?

### **Step 3: Creating a Pie Chart for Sales Distribution**

A pie chart shows the contribution of different products to total sales.

1. Click on Pie Chart from the Visualizations Pane
2. Drag Product to the Legend
3. Drag Sales to the Values

#### **Insights from Pie Chart:**

- Which products contribute most to sales?
- Which product has the least sales?

### **Step 4: Creating a Line Chart for Sales Trend**

A line chart helps track monthly sales trends.

1. Click on Line Chart from the Visualizations Pane
2. Drag Date to the X-axis
3. Drag Sales to the Y-axis
4. Set the date hierarchy to view sales by Month & Year

#### **Insights from Line Chart:**

- When were sales highest?
- Identify seasonal trends or sales dips

### **Step 5: Adding Conditional Formatting**

Conditional formatting highlights important data points.

1. Click on the Bar Chart
2. Select the Sales field
3. Go to Format Pane → Data Colors → Conditional Formatting
4. Apply gradient coloring (e.g., red for low sales, green for high sales)

#### **Benefit of Conditional Formatting:**

- Quickly spot high and low-performing categories

## **Step 6: Using Drill-Through for Detailed Insights**

Drill-through allows users to click on a data point to see detailed insights.

1. Create a new page
2. Add a Table Visualization with Product, Sales, and Profit
3. Enable Drill-Through on Product Category
4. Now, clicking a category in the bar chart will filter the detailed sales table

### **Benefit of Drill-Through:**

- Users can dive deeper into specific product sales

## **Step 7: Creating Custom Tooltips**

Tooltips provide additional on-hover insights.

1. Click on Format Pane
2. Enable Tooltips
3. Add fields such as Sales, Profit, Discount to display inside the tooltip

### **Benefit of Custom Tooltips:**

- Provides extra context without taking up space on the dashboard

## **Step 8: Implementing Slicers (Filters)**

Slicers allow users to filter data dynamically.

1. Click on Slicer from the Visualizations Pane
2. Drag Region into the slicer
3. Users can now select different regions to filter sales data

### **Benefit of Slicers:**

- Allows interactive filtering based on user selections

## 4. Summary of What We Built

- ✓ Bar Chart → Product category-wise sales
- ✓ Pie Chart → Product sales contribution
- ✓ Line Chart → Sales trend over time
- ✓ Conditional Formatting → Highlights top-performing categories
- ✓ Drill-Through → Clicking a category shows detailed data
- ✓ Custom Tooltips → Displays sales & profit on hover
- ✓ Slicers → Enables users to filter sales by region

## 5. Conclusion

Power BI visualizations make data storytelling easy and enable better decision-making. By combining charts, drill-throughs, conditional formatting, and slicers, we can create powerful dashboards for business insights.

### Mini Project 1: Sales Performance Dashboard in Power BI

#### Project Objective:

Create an interactive Sales Performance Dashboard using Power BI to analyze sales trends, regional performance, and top-selling products using various charts, conditional formatting, drill-through, and slicers.

#### Step-by-Step Implementation

##### Step 1: Import Data into Power BI

1. Download Sample Dataset: Use an Excel or CSV file with sales data.
2. Open Power BI Desktop → Click on Home → Get Data → Excel.
3. Select the file (e.g., sales\_data.xlsx) and click Load.

**Dataset Columns:**

Order ID	Date	Product	Category	Sales	Profit	Quantity	Region
1001	2023-01-10	Laptop	Electronics	1000	200	2	New York
1002	2023-01-15	Phone	Electronics	800	100	1	California
1003	2023-02-01	Chair	Furniture	150	50	4	Texas
...	...	...	...	...	...	...	...

**Step 2: Creating a Bar Chart for Sales by Category**

1. Click on Clustered Bar Chart from the Visualizations Pane.
2. Drag Category to X-axis.
3. Drag Sales to Y-axis.
4. Format the chart:
  - a. Change colors based on sales values.
  - b. Add Data Labels for better readability.
  - c. Adjust the title and axis names.

**Insights:**

- Which product category has the highest sales?
- Compare sales performance across categories.

**Step 3: Creating a Pie Chart for Sales Distribution by Region**

1. Click on Pie Chart from the Visualizations Pane.
2. Drag Region into the Legend.
3. Drag Sales into the Values field.
4. Adjust the colors and labels.

**Insights:**

- Which region contributes the most to sales?
- Identify low-performing regions.

#### **Step 4: Creating a Line Chart for Sales Trend Over Time**

1. Click on Line Chart.
2. Drag Date to the X-axis.
3. Drag Sales to the Y-axis.
4. Change Date Field Hierarchy to display Monthly Sales.
5. Format the line style and color.

#### **Insights:**

- Identify seasonal trends in sales.
- Spot sales spikes and drops over time.

#### **Step 5: Applying Conditional Formatting**

1. Click on the Bar Chart.
2. Go to Format Pane → Data Colors → Conditional Formatting.
3. Set Color Scale:
  - a. Red for low sales
  - b. Yellow for moderate sales
  - c. Green for high sales

#### **Insights:**

- Quickly identify high and low-performing categories.

#### **Step 6: Implementing Drill-Through for Product Details**

1. Create a new page and rename it "Product Details".
2. Add a Table Visualization with Product, Sales, and Profit.
3. Enable Drill-Through:
  - a. Click on the Page → Add Category as a Drill-through field.
  - b. Now, clicking a Category in the Bar Chart will show detailed product sales.

#### **Insights:**

- Click a category to view all its products and profits.

### **Step 7: Adding Custom Tooltips**

1. Click on Line Chart.
2. Enable Tooltips under the Format Pane.
3. Add Profit and Quantity Sold as tooltip fields.

#### **Benefit:**

- Hover over the sales trend to see extra details.

### **Step 8: Using Slicers for Interactive Filtering**

1. Click on Slicer from the Visualizations Pane.
2. Drag Region into the slicer.
3. Drag Category into another slicer.
4. Users can now filter sales data dynamically.

#### **Insights:**

- Compare sales between different regions and categories.

### **Step 9: Publish and Share**

1. Click Publish to upload the report to Power BI Service.
2. Share the interactive dashboard with the team.

#### **Final Dashboard Includes:**

- ✓ Bar Chart → Sales by Category
- ✓ Pie Chart → Sales Distribution by Region
- ✓ Line Chart → Monthly Sales Trend
- ✓ Conditional Formatting → Highlighting top-performing categories
- ✓ Drill-Through → Clicking a category shows product-level details
- ✓ Custom Tooltips → Extra insights on hover
- ✓ Slicers → Interactive filtering

## Mini Project 2: Customer Purchase Behavior Analysis

### Project Objective:

Analyze customer purchase behavior using different Power BI visualizations to track buying trends, product popularity, and regional customer preferences.

### Step-by-Step Implementation

#### Step 1: Import Customer Data

1. Open Power BI Desktop.
2. Click on Get Data → Excel.
3. Load the customer\_purchase.xlsx file.

#### Dataset Columns:

Customer ID	Date	Product	Category	Purchase Amount	Age Group	Region
C001	2023-01-05	Phone	Electronics	700	25-34	New York
C002	2023-02-10	Laptop	Electronics	1200	35-44	Texas
C003	2023-03-15	Sofa	Furniture	800	45-54	California
...	...	...	...	...	...	...

#### Step 2: Create a Column Chart for Age Group vs Purchase Amount

1. Select Clustered Column Chart.
2. Drag Age Group to the X-axis.
3. Drag Purchase Amount to the Y-axis.
4. Format the colors, labels, and title.

**Insights:**

- Which age group spends the most?

**Step 3: Create a Pie Chart for Product Category Distribution**

1. Select Pie Chart.
2. Drag Category into Legend.
3. Drag Purchase Amount into Values.

**Insights:**

- Identify the most popular product category.

**Step 4: Create a Map Chart for Customer Purchases by Region**

1. Click on Map Visualization.
2. Drag Region to Location.
3. Drag Purchase Amount to Values.

**Insights:**

- Which region has the highest spending?

**Step 5: Apply Conditional Formatting on Purchase Amount**

1. Click on the Column Chart.
2. Enable Conditional Formatting in Data Colors.
3. Set a gradient from blue (low spenders) to red (high spenders).

**Insights:**

- Quickly identify high-spending age groups.

**Step 6: Use Drill-Through for Individual Customer Details**

1. Create a new page → Add a Table.
2. Enable Drill-Through on Customer ID.
3. Clicking an age group now filters purchases by individual customers.

1329

**Private & Confidential : Vetri Technology Solutions**

**Benefit:**

- Understand individual customer spending behavior.

**Step 7: Use Slicers for Interactive Analysis**

1. Add Slicers for:
  - a. Region
  - b. Category
2. Users can filter data dynamically.

**Insights:**

- Compare spending trends for different regions and product types.

**Step 8: Publish & Share**

1. Click Publish in Power BI.
2. Share with marketing and sales teams.

**Final Dashboard Includes:**

- ✓ Column Chart → Age Group vs Purchase Amount
- ✓ Pie Chart → Purchase Distribution by Category
- ✓ Map Chart → Regional Spending Distribution
- ✓ Conditional Formatting → Highlighting top-spending groups
- ✓ Drill-Through → Customer-specific purchase details
- ✓ Slicers → Interactive filtering

**Conclusion**

These Power BI mini-projects demonstrate real-world business analysis, helping users to visualize sales trends, customer spending patterns, and regional performance using charts, drill-through, slicers, and tooltips.

## Mini Project: Employee Performance & Attendance Dashboard in Power BI

### Project Objective:

Create an interactive Employee Performance & Attendance Dashboard in Power BI to analyze employee productivity, attendance trends, department-wise performance, and salary distribution using various Power BI visualization techniques.

### Day 90 Tasks

#### 1. Import Employee Data into Power BI

- Load the dataset from an Excel or CSV file.

#### 2. Create a Bar Chart for Department-Wise Performance

- Show average performance ratings per department.

#### 3. Build a Pie Chart for Attendance Distribution

- Display the percentage of present vs. absent employees.

#### 4. Develop a Line Chart for Monthly Attendance Trends

- Analyze employee attendance trends over months.

#### 5. Create a Map Visualization for Employee Locations

- Show the geographical distribution of employees.

#### 6. Apply Conditional Formatting on Salary Distribution

- Highlight high and low salary ranges using color formatting.

#### 7. Use Drill-Through to Show Employee Details

- Clicking a department should reveal employee-specific performance and salary details.

#### 8. Design Custom Tooltips for Performance Metrics

- When hovering over performance ratings, show additional metrics like overtime hours worked.

#### 9. Implement Hierarchies for Employee Data

- Create hierarchical filters for Employee → Department → Job Role.

#### 10. Use Slicers for Interactive Filtering

- Allow users to filter data by department, job role, and month.

#### 11. Create a Card Visualization for Key Metrics

- Show total employees, average attendance rate, and highest salary.

#### 12. Build a KPI Visualization for Productivity Trends

- Track monthly changes in performance ratings.

#### 13. Publish & Share the Dashboard

- Upload the report to Power BI Service and enable sharing.

#### **Outcome:**

This dashboard will help HR teams and managers to track employee performance, monitor attendance, and make data-driven decisions based on visual insights.

## Mini Project 1: Sales Performance Dashboard

### Objective:

Create a Sales Performance Dashboard in Power BI to track revenue, sales trends, top-selling products, and regional performance using various visualization techniques.

### Key Features:

- Import sales data (Excel, CSV, or SQL).
- Create bar charts for product-wise sales comparison.
- Use line charts to track monthly/quarterly revenue trends.
- Implement geo-maps to analyze sales by region.
- Apply conditional formatting to highlight best-selling products.
- Add drill-through functionality for detailed product-level insights.
- Use slicers for dynamic filtering by product, region, and time period.

## Mini Project 2: Customer Feedback Analysis Dashboard

### Objective:

Build a Customer Feedback Analysis Dashboard to analyze customer sentiment, ratings, and feedback trends using Power BI visualizations.

### Key Features:

- Import customer feedback dataset (Excel, Web API, or SQL).
- Create pie charts for sentiment analysis (Positive, Neutral, Negative).
- Use bar charts to show average ratings by product/service.
- Implement word cloud visualization for most common feedback words.
- Apply conditional formatting to highlight critical issues.
- Use custom tooltips to show detailed feedback when hovering over reviews.
- Implement hierarchies to filter data by product category, region, or customer type.

# Day 91

## Advanced DAX & Time Intelligence in Power BI

### What is Time Intelligence in Power BI?

Time Intelligence in Power BI refers to DAX (Data Analysis Expressions) functions that allow users to analyze time-based data effectively. These functions help businesses track performance, compare trends, and generate cumulative summaries.

### Why Use Time Intelligence?

- Compare sales performance across different time periods (Year-over-Year, Month-over-Month).
- Calculate cumulative totals (Running Total, Year-to-Date (YTD), Quarter-to-Date (QTD)).
- Compute moving averages for trend analysis.
- Analyze historical data dynamically.

### Common DAX Time Intelligence Functions

#### 1. DATEADD()

**Definition:** Shifts dates forward or backward by a given period.

#### Syntax:

DATEADD(<dates>, <number\_of\_intervals>, <interval>)

- <dates> → The column containing date values.
- <number\_of\_intervals> → Positive (future) or negative (past).
- <interval> → "DAY", "MONTH", "QUARTER", "YEAR".

### **Example: Calculate previous month's sales**

Previous\_Month\_Sales = CALCULATE(SUM(Sales[Total\_Sales]),  
DATEADD(Sales[Date], -1, MONTH))

#### **What This Does?**

- Shifts the Date column one month back.
- Calculates the total sales for the previous month.

## **2. TOTALYTD()**

**Definition:** Calculates Year-To-Date (YTD) totals.

#### **Syntax:**

TOTALYTD(<expression>, <dates>[, <filter>])

- <expression> → The calculation (e.g., SUM of sales).
- <dates> → The column containing dates.
- <filter> → (Optional) Additional filter.

### **Example: Calculate Year-To-Date revenue**

Sales\_YTD = TOTALYTD(SUM(Sales[Total\_Sales]), Sales[Date])

#### **What This Does?**

- Computes cumulative total sales from January 1st to today.

## **3. PREVIOUSMONTH()**

**Definition:** Retrieves data for the previous month.

#### **Syntax:**

PREVIOUSMONTH(<dates>)

**Example: Get total sales for the previous month**

```
Previous_Month_Sales = CALCULATE(SUM(Sales[Total_Sales]),
PREVIOUSMONTH(Sales[Date]))
```

**What This Does?**

- Filters the Sales[Date] column to return the previous month's data.

**4. SAMEPERIODLASTYEAR()**

**Definition:** Returns the same period from the previous year.

**Syntax:**

```
SAMEPERIODLASTYEAR(<dates>)
```

**Example: Compare total revenue for the same period last year**

```
Last_Year_Sales = CALCULATE(SUM(Sales[Total_Sales]),
SAMEPERIODLASTYEAR(Sales[Date]))
```

**What This Does?**

- Fetches sales data for the same period from last year.

**5. Running Totals & Moving Averages****Running Total (Cumulative Sales YTD):**

```
Running_Total = CALCULATE(SUM(Sales[Total_Sales]), FILTER(ALL(Sales[Date]),
Sales[Date] <= MAX(Sales[Date])))
```

**What This Does?**

- Uses FILTER() to select all previous dates.
- SUM() calculates cumulative sales up to today.

**3-Month Moving Average:**

```
Moving_Avg_3M = AVERAGEX(DATESINPERIOD(Sales[Date], MAX(Sales[Date]), -3,
MONTH), SUM(Sales[Total_Sales]))
```

## What This Does?

- DATESINPERIOD() selects last 3 months.
- AVERAGEX() computes the average sales for that period.

## 6. Using VAR and RETURN in DAX

### Definition:

VAR defines temporary variables to improve readability and performance.

### Syntax:

```
VAR variable_name = expression
```

```
RETURN
```

```
result_expression
```

### Example: Calculate Revenue Growth Percentage

```
VAR CurrentMonth = SUM(Sales[Total_Sales])
```

```
VAR PreviousMonth = CALCULATE(SUM(Sales[Total_Sales]),  
PREVIOUSMONTH(Sales[Date]))
```

```
RETURN
```

```
IF(PreviousMonth = 0, BLANK(), (CurrentMonth - PreviousMonth) /  
PreviousMonth * 100)
```

## What This Does?

- VAR stores Current Month's Sales and Previous Month's Sales.
- The formula calculates percentage growth.
- If PreviousMonth = 0, it returns BLANK() to avoid division errors.

## Implementation: Real-Life Example

### Scenario: Sales Performance Dashboard

**Objective:** Create a Power BI report to analyze monthly sales trends, YTD sales, and previous year comparisons.

#### Step 1: Load Data

- Import sales data into Power BI.
- Ensure a Date column exists.

#### Step 2: Create Measures in Power BI

##### 1. Total Sales

Total\_Sales = SUM(Sales[Total\_Sales])

##### 2. Previous Month Sales

Previous\_Month\_Sales = CALCULATE(SUM(Sales[Total\_Sales]),  
PREVIOUSMONTH(Sales[Date]))

##### 3. Sales Year-To-Date (YTD)

Sales\_YTD = TOTALYTD(SUM(Sales[Total\_Sales]), Sales[Date])

##### 4. Same Period Last Year

Last\_Year\_Sales = CALCULATE(SUM(Sales[Total\_Sales]),  
SAMEPERIODLASTYEAR(Sales[Date]))

##### 5. Revenue Growth Percentage

VAR CurrentMonth = SUM(Sales[Total\_Sales])

VAR PreviousMonth = CALCULATE(SUM(Sales[Total\_Sales]),

```
PREVIOUSMONTH(Sales[Date]))
```

```
RETURN
```

```
IF(PreviousMonth = 0, BLANK(), (CurrentMonth - PreviousMonth) /  
PreviousMonth * 100)
```

### **Step 3: Visualize in Power BI**

- Bar Chart: Monthly Sales vs Previous Month Sales.
- Line Chart: YTD Sales Trend.
- Card Visualization: Revenue Growth %.
- Slicer: Year & Month filter.

### **Conclusion**

- DATEADD() → Shifts dates for comparisons.
- TOTALYTD() → Cumulative sales for the year.
- PREVIOUSMONTH() → Retrieves last month's sales.
- SAMEPERIODLASTYEAR() → Compares with last year.
- Running Totals & Moving Averages → Analyze trends.
- VAR & RETURN → Optimize complex calculations.

### **Key Takeaways**

- ✓ Use Time Intelligence functions for trend analysis.
- ✓ Combine DAX functions to create advanced reports.
- ✓ Use variables (VAR) for better performance.

## Mini Project 1: Sales Performance Analysis using Time Intelligence in Power BI

### Project Objective:

Analyze sales trends over time by using DAX Time Intelligence functions in Power BI. The goal is to track monthly sales, compare sales with the previous year, calculate running totals, and generate moving averages.

### Step 1: Dataset Preparation

- Create a sample Sales Data table with the following fields:
  - OrderID: Unique order identifier.
  - Date: Order date.
  - Total\_Sales: Total amount of the sale.
  - Category: Product category.

### Sample Data (CSV/Excel format):

OrderID	Date	Total_Sales	Category
101	2023-01-15	500	Electronics
102	2023-02-10	700	Clothing
103	2023-02-20	450	Electronics
104	2023-03-05	800	Furniture
105	2023-03-20	600	Electronics
106	2024-01-10	900	Clothing
107	2024-02-05	750	Furniture
108	2024-03-15	1100	Electronics

### Step 2: Load Data into Power BI

1. Open Power BI Desktop.
2. Click Home > Get Data > Excel (or CSV).
3. Select the Sales Data file and click Load.
4. Ensure the Date column is recognized as a Date type.

### **Step 3: Create DAX Measures**

#### **1. Total Sales Measure**

Total\_Sales = SUM(Sales[Total\_Sales])

This calculates the sum of total sales.

#### **2. Previous Month Sales**

Previous\_Month\_Sales = CALCULATE(SUM(Sales[Total\_Sales]),  
PREVIOUSMONTH(Sales[Date]))

Retrieves the sales data from the previous month.

#### **3. Year-To-Date (YTD) Sales**

Sales\_YTD = TOTALYTD(SUM(Sales[Total\_Sales]), Sales[Date])

Computes cumulative total sales for the year.

#### **4. Same Period Last Year (YoY Comparison)**

Last\_Year\_Sales = CALCULATE(SUM(Sales[Total\_Sales]),  
SAMEPERIODLASTYEAR(Sales[Date]))

Compares this year's sales with the same period last year.

#### **5. Running Total (Cumulative Sales)**

```
Running_Total =
CALCULATE(
    SUM(Sales[Total_Sales]),
    FILTER(ALL(Sales[Date]), Sales[Date] <= MAX(Sales[Date]))
)
```

Computes cumulative sales up to the current date.

## 6. 3-Month Moving Average

```
Moving_Avg_3M =  
AVERAGEX(  
    DATESINPERIOD(Sales[Date], MAX(Sales[Date]), -3, MONTH),  
    SUM(Sales[Total_Sales])  
)
```

Calculates the average sales for the last 3 months.

## Step 4: Create Visuals

### 1. Total Sales Bar Chart

- a. Select Bar Chart.
- b. Drag Date to X-axis and Total\_Sales to Y-axis.

### 2. Previous Month vs Current Month Comparison

- a. Select Line Chart.
- b. Add Total Sales and Previous Month Sales.

### 3. Year-over-Year (YoY) Sales Comparison

- a. Create a Line Chart.
- b. Add Total Sales and Last Year Sales.

### 4. Running Total Chart

- a. Select Area Chart.
- b. Add Running Total.

### 5. Moving Average Trend

- a. Use a Line Chart.
- b. Add Moving\_Avg\_3M.

### Step 5: Insights from the Report

- Compare monthly sales trends and track revenue growth.
- Identify seasonal patterns using YoY comparison.
- Use running totals to track sales accumulation over time.
- Use moving averages to smooth fluctuations.

## Mini Project 2: Employee Salary Growth Analysis Using Time Intelligence in Power BI

### Project Objective:

Analyze salary trends over time by using DAX Time Intelligence functions in Power BI. The goal is to track monthly salary expenditures, compare salaries with the previous year, and calculate moving averages.

### Step 1: Dataset Preparation

- Create a sample Employee Salary Data table with the following fields:
  - EmployeeID: Unique employee identifier.
  - Date: Salary payment date.
  - Department: Employee's department.
  - Salary\_Paid: Monthly salary amount.

### Sample Data (CSV/Excel format):

EmployeeID	Date	Salary_Paid	Department
201	2023-01-01	5000	HR
202	2023-02-01	5200	IT
203	2023-03-01	5400	Sales
204	2023-04-01	5800	IT
205	2023-05-01	6000	HR
206	2024-01-01	6200	Sales
207	2024-02-01	6500	HR
208	2024-03-01	6800	IT

## Step 2: Load Data into Power BI

1. Open Power BI Desktop.
2. Click Home > Get Data > Excel (or CSV).
3. Select the Employee Salary Data file and click Load.
4. Ensure the Date column is recognized as a Date type.

## Step 3: Create DAX Measures

### 1. Total Salary Paid

Total\_Salary = SUM(Salary[Salary\_Paid])

Calculates the total salary paid.

### 2. Previous Month Salary

Previous\_Month\_Salary = CALCULATE(SUM(Salary[Salary\_Paid]),  
PREVIOUSMONTH(Salary[Date]))

Retrieves salaries paid last month.

### 3. Year-To-Date (YTD) Salary

Salary\_YTD = TOTALYTD(SUM(Salary[Salary\_Paid]), Salary[Date])

Computes cumulative salary payments for the year.

### 4. Last Year Salary (YoY Comparison)

Last\_Year\_Salary = CALCULATE(SUM(Salary[Salary\_Paid]),  
SAMEPERIODLASTYEAR(Salary[Date]))

Compares this year's salary payments with last year's.

## 5. Running Total Salary

```
Running_Total_Salary =  
CALCULATE(  
    SUM(Salary[Salary_Paid]),  
    FILTER(ALL(Salary[Date]), Salary[Date] <= MAX(Salary[Date]))  
)
```

Computes cumulative salaries paid to date.

## 6. Moving Average for Salary Trend

```
Moving_Avg_Salary =  
AVERAGEX(  
    DATESINPERIOD(Salary[Date], MAX(Salary[Date]), -3, MONTH),  
    SUM(Salary[Salary_Paid]))  
)
```

Calculates salary trend using a 3-month moving average.

### Step 4: Create Visuals

1. Total Salary Paid Line Chart
2. Previous Month vs Current Month Salary Comparison
3. Year-over-Year (YoY) Salary Comparison
4. Running Total Salary Chart
5. Moving Average Salary Trend

### Step 5: Insights from the Report

- Track monthly salary growth trends.
- Identify salary changes between departments.
- Use running totals to analyze total salary payout.
- Forecast salary budgets using moving averages.

## Mini Project: Sales and Revenue Forecasting using Advanced DAX & Time Intelligence

### Objective:

Analyze and forecast monthly sales and revenue trends using Advanced DAX Time Intelligence functions in Power BI. This project will include Year-over-Year (YoY) comparison, running totals, moving averages, and custom calculations to help business stakeholders make data-driven decisions.

### Day 91 Tasks:

#### 1. Load Sales Data into Power BI

- Import a dataset containing sales transactions with fields: OrderID, Date, Total\_Sales, Customer\_Segment, and Category.

#### 2. Create a Date Table for Time Intelligence Functions

- Generate a date table using DAX to enable proper time-based calculations.

#### 3. Calculate Total Sales Using SUM()

- Create a DAX measure to sum up the total sales for analysis.

#### 4. Compare Monthly Sales with Previous Month Sales Using PREVIOUSMONTH()

- Implement a measure to fetch and compare the total sales from the last month.

#### 5. Calculate Year-To-Date (YTD) Sales Using TOTALYTD()

- Track cumulative sales from the start of the year up to the current date.

#### 6. Perform a Year-over-Year (YoY) Comparison Using SAMEPERIODLASTYEAR()

- Create a measure to compare the sales from the same period in the previous year.

7. Compute Running Totals (Cumulative Sales) Using CALCULATE() & FILTER()

- Implement a running total measure to track sales accumulation over time.

8. Generate a Moving Average Trend for Sales Using AVERAGEX()

- Compute a 3-month moving average to smooth sales fluctuations.

9. Use DATEADD() to Compare Sales for the Last 6 Months

- Create a measure that shifts the date backward by 6 months and retrieves sales data.

10. Use VAR and RETURN Statements to Optimize a Custom Sales Calculation

- Define variables in DAX to enhance readability and performance of a complex calculation.

11. Apply Filters Using CALCULATE() & FILTER() to Analyze Specific Customer Segments

- Create a measure that calculates total sales for a selected customer segment.

12. Create an Interactive Dashboard with Line Charts and Bar Charts

- Visualize monthly trends, YoY comparisons, and running totals using different charts.

13. Add Slicers for Dynamic Filtering by Year, Month, and Category

- Implement slicers to allow users to dynamically filter sales trends by year, month, and category.

## Mini Project 1: E-commerce Sales Performance Analysis using Advanced DAX & Time Intelligence

### Objective:

Analyze e-commerce sales performance by implementing time intelligence functions to track revenue trends, seasonal patterns, and customer behavior.

### Requirements:

- Import an E-commerce Sales dataset with fields like OrderID, Order Date, Customer ID, Product Category, and Total Sales.
- Create a date table to enable time-based calculations.
- Implement TOTALYTD() to calculate Year-to-Date (YTD) revenue.
- Use PREVIOUSMONTH() and SAMEPERIODLASTYEAR() to compare sales performance across different time periods.
- Compute running totals and moving averages to identify long-term sales trends.
- Use VAR and RETURN statements to optimize complex DAX calculations.
- Build interactive charts and slicers for category-wise and region-wise sales comparisons.

## Mini Project 2: Financial KPI Dashboard for Monthly Budget vs.

### Actuals

### Objective:

Develop a Power BI financial dashboard to track budget vs. actual expenses for a company and analyze financial performance using Advanced DAX.

### Requirements:

- Import a dataset containing Budgeted vs. Actual Expenses with fields: Month, Category, Budgeted Amount, and Actual Amount.
- Use DAX measures to calculate the budget variance and highlight over/under-spending.

- Implement DATEADD() to compare actual spending over different time periods.
- Create TOTALYTD() measures to track year-to-date financial performance.
- Use FILTER() and CALCULATE() to analyze expenses by department and category.
- Build dynamic bar charts, variance indicators, and financial KPIs using Power BI.
- Implement slicers to allow users to filter expenses by year, category, and department.

## Day 92

### Power BI Service & Sharing Reports

Power BI Service is a cloud-based platform that allows users to publish, share, and collaborate on Power BI reports and dashboards. It helps in distributing reports to different stakeholders while ensuring security and controlled access.

#### 1. Publishing Reports to Power BI Service

##### Definition

Publishing reports to Power BI Service allows users to share and access reports from anywhere using a web browser.

##### Steps to Publish a Report

1. Create a Report in Power BI Desktop
  - a. Build a report using Power BI Desktop.
2. Sign in to Power BI Service
  - a. Click on "Sign In" under the Power BI ribbon.
3. Publish the Report

- a. Click on File → Publish → Publish to Power BI Service.
  - b. Select a workspace (e.g., "My Workspace").
4. Access the Report in Power BI Service
    - a. Go to Power BI Service.
    - b. Find your report under Workspaces.

### **Real-life Example**

A sales manager wants to track monthly sales. The report is created in Power BI Desktop and published to Power BI Service, allowing the manager to access it online and share it with their team.

## **2. Creating Dashboards & Tiles**

### **Definition**

A dashboard is a collection of visualizations (tiles) that provide a high-level view of data.

### **Steps to Create a Dashboard**

1. Open Power BI Service → Navigate to your report.
2. Pin Visuals to a Dashboard
  - a. Click on a visualization → Select Pin to Dashboard.
  - b. Choose an existing dashboard or create a new one.
3. Customize the Dashboard
  - a. Rearrange, resize, or update tiles as needed.

### **Real-life Example**

A CEO wants to see an overview of company revenue, profit, and top-selling products on a single page. A Power BI dashboard is created, summarizing key metrics.

### 3. Using Workspaces & Apps for Collaboration

#### Definition

- Workspaces: Shared areas where teams collaborate on reports.
- Apps: Packaged versions of reports & dashboards shared with users.

#### Steps to Use Workspaces & Apps

1. Create a Workspace
  - a. Click on Workspaces → Create a new workspace.
2. Add Members & Assign Roles
  - a. Add viewers, contributors, admins.
3. Publish an App
  - a. After building reports, click "Create App" to share them securely.

#### Real-life Example

An HR team uses a workspace to analyze employee performance, and then shares the final HR reports via an app to senior management.

### 4. Setting Up Row-Level Security (RLS)

#### Definition

RLS restricts data access so that users only see data relevant to them.

#### Steps to Implement RLS

1. Create a Role in Power BI Desktop
  - a. Go to Modeling → Manage Roles → Create New Role.
  - b. Apply a DAX filter, e.g., [Region] = "North" for regional managers.
2. Assign Users to the Role in Power BI Service
  - a. Open the dataset in Power BI Service → Security → Assign users to roles.

#### Real-life Example

A sales team should only see sales data for their assigned region. RLS ensures that a sales manager in New York does not see sales data from California.

## 5. Embedding Power BI Reports in Web & SharePoint

### Definition

Power BI reports can be embedded in web applications, SharePoint, or third-party platforms for easy access.

### Steps to Embed a Report

1. Go to Power BI Service → Open a Report
2. Click on "File" → "Embed Report"
  - a. Choose "Website or Portal" for public sharing.
  - b. Choose "SharePoint Online" for internal company sharing.
3. Copy the Embed Link & Use It
  - a. For SharePoint: Insert the link into a SharePoint Power BI web part.
  - b. For Websites: Use an iFrame to embed the report.

### Real-life Example

A university admin wants to display student performance dashboards on their intranet portal. The report is embedded in SharePoint, allowing faculty members to view it securely.

### Summary Table

Feature	Description	Real-Life Example
<b>Publishing Reports</b>	Upload reports to Power BI Service	Sales reports accessible online
<b>Dashboards &amp; Tiles</b>	Interactive, high-level views	CEO dashboard for financial KPIs
<b>Workspaces &amp; Apps</b>	Team collaboration on reports	HR team managing employee reports
<b>Row-Level Security (RLS)</b>	Restricting data access by role	Regional managers see only their region's data
<b>Embedding Reports</b>	Share reports on web, SharePoint	University shares dashboards on the intranet

## Project 1: Sales Performance Dashboard for a Retail Chain

### Objective

The company wants to track sales performance across multiple stores and share it securely with regional managers. Each manager should see only their assigned region's data using Row-Level Security (RLS).

### Step-by-Step Implementation

#### Step 1: Create a Power BI Report (Sales Report)

1. Load the Sales Data
  - a. Open Power BI Desktop.
  - b. Click on Home → Get Data → Excel/SQL Server.
  - c. Load the following tables:
    - i. SalesData: Contains sales transactions.
    - ii. Stores: Contains store details and region.
    - iii. Employees: Contains sales managers assigned to regions.
2. Create Basic Visualizations
  - a. Sales Trend → Line chart for monthly sales.
  - b. Top 5 Stores by Revenue → Bar chart.
  - c. Total Sales → Card visualization.
  - d. Sales by Region → Map visualization.

#### Step 2: Publish Report to Power BI Service

1. Click File → Publish → Publish to Power BI Service.
2. Select the workspace "Retail Sales Workspace".
3. After publishing, open Power BI Service to access the report.

#### Step 3: Create a Dashboard & Pin Tiles

1. In Power BI Service, open the Sales Report.
2. Click "Pin to Dashboard" for each visualization.

3. Select "New Dashboard" → Name it "Sales Performance Dashboard".
4. Arrange tiles to make the dashboard user-friendly.

#### **Step 4: Configure Row-Level Security (RLS)**

##### **1. Define Roles in Power BI Desktop**

1. Go to Modeling → Manage Roles.
2. Click Create New Role → Name it RegionalManager.
3. Apply a filter to restrict data by region using DAX filter: [Region] =  
USERPRINCIPALNAME()
4. Save and publish the updated report.

##### **2. Assign Users in Power BI Service**

1. In Power BI Service, open the dataset → Click Security.
2. Assign users to their roles based on email addresses.

#### **Step 5: Share Report Using Workspaces & Apps**

1. Create a Workspace → "Retail Sales Workspace".
2. Add users:
  - a. Admin: Full access.
  - b. Regional Managers: View only their region.
3. Publish an App with selected reports.
4. Share the App URL with stakeholders.

#### **Step 6: Embed Report in SharePoint**

1. Open the report in Power BI Service.
2. Click File → Embed Report → SharePoint Online.

3. Copy the embed link and add it to a SharePoint page using the Power BI web part.

### **Expected Outcome**

- ✓ Regional managers can view sales only for their assigned region.
- ✓ Executives get a real-time dashboard.
- ✓ Reports are securely embedded in SharePoint for easy access.

## **Project 2: Employee Performance Dashboard for HR Department**

### **Objective**

The HR department wants a dashboard to analyze employee performance and share it with different managers while ensuring data security using RLS.

### **Step-by-Step Implementation**

#### **Step 1: Load Employee Data in Power BI**

1. Open Power BI Desktop.
2. Import data from HR Database (SQL Server).
3. Load the following tables:
  - a. EmployeeDetails: Employee name, department, manager.
  - b. PerformanceScores: Monthly performance ratings.
  - c. HRMetrics: Attendance, promotions, feedback.

#### **Step 2: Create Visualizations**

1. Employee Performance Trend → Line Chart.
2. Top 10 Employees by Score → Bar Chart.
3. Attendance Rate by Department → Pie Chart.
4. Employee Count per Department → Card Visualization.

### **Step 3: Publish the Report to Power BI Service**

1. Click File → Publish to Power BI Service.
2. Select the "HR Analytics Workspace".
3. Open Power BI Service to access the report.

### **Step 4: Create & Customize Dashboard**

1. Open the report → Click Pin to Dashboard for each visualization.
2. Name the new dashboard "Employee Performance Dashboard".
3. Arrange tiles for better readability.

### **Step 5: Implement Row-Level Security (RLS)**

#### **1. Define Roles in Power BI Desktop**

1. Go to Modeling → Manage Roles.
2. Create a role named "Manager".
3. Apply the following DAX filter to ensure managers see only their department's data: [Department] =  
LOOKUPVALUE(Employees[Department], Employees[Email],  
USERPRINCIPALNAME())
4. Save & publish the updated dataset.

#### **2. Assign Roles in Power BI Service**

1. Open Power BI Service → Go to dataset Security.
2. Assign department managers to the Manager role.

### **Step 6: Share Reports Using Workspaces & Apps**

1. Create a Workspace → "HR Analytics Workspace".

2. Add HR Managers & Department Heads with appropriate permissions.
3. Publish the HR App and share it with department managers.

### **Step 7: Embed Report in SharePoint & Web**

1. Open the report in Power BI Service.
2. Click File → Embed Report → SharePoint Online.
3. Copy the link and paste it into a SharePoint page for HR team access.

### **Expected Outcome**

- ✓ HR team & managers can track employee performance securely.
- ✓ Department heads can only see employees in their department.
- ✓ Reports are shared via Power BI App & embedded in SharePoint.

### **Final Summary**

<b>Feature</b>	<b>Project 1: Sales Performance</b>	<b>Project 2: Employee Performance</b>
<b>Publishing Reports</b>	Sales report published in Power BI Service	Employee report published in Power BI Service
<b>Creating Dashboards</b>	Sales dashboard with KPIs	Employee performance dashboard
<b>Using Workspaces &amp; Apps</b>	Retail workspace for sales managers	HR workspace for department managers
<b>Row-Level Security (RLS)</b>	Managers see only their assigned region	Managers see only their department's employees
<b>Embedding Reports</b>	Sales dashboard embedded in SharePoint	HR reports embedded in SharePoint

## Real-Life Mini Project: Customer Support Performance Dashboard

### Objective:

A company wants to track customer support performance using Power BI Service. The Customer Support Manager should see an overview, while each Team Lead should only see their team's performance using Row-Level Security (RLS). The dashboard should be shared securely via Power BI Apps and SharePoint.

### Day 92 Tasks

#### Task 1: Load Data into Power BI

- Import data from an Excel file / SQL Database containing:
  - TicketsData (Ticket ID, Created Date, Resolved Date, Status, Assigned Team, Agent, Resolution Time).
  - Teams (Team ID, Team Name, Team Lead Email).
  - Agents (Agent ID, Name, Team ID).

#### Task 2: Clean & Transform Data

- Ensure correct date formats for ticket tracking.
- Remove duplicate or missing records.
- Create relationships between tables (TicketsData ↔ Teams ↔ Agents).

#### Task 3: Create Key Visualizations

- Number of Tickets Resolved per Month (Line Chart).
- Average Resolution Time per Team (Bar Chart).
- Pending vs Resolved Tickets (Pie Chart).
- Top Performing Agents by Resolved Tickets (Table).

#### Task 4: Apply Conditional Formatting

- Highlight tickets resolved within SLA in green.

- Highlight tickets breaching SLA in red.

#### Task 5: Create & Publish Report to Power BI Service

- Publish the report to Power BI Service → Customer Support Workspace.

#### Task 6: Create an Interactive Dashboard

- Pin key visualizations as tiles.
- Add a KPI tile to display total tickets resolved.
- Arrange tiles logically for easy navigation.

#### Task 7: Configure Row-Level Security (RLS)

- Restrict Team Leads to view only their team's performance using DAX filters.
- Test security settings with different users.

#### Task 8: Set Up a Power BI Workspace for Collaboration

- Create "Customer Support Workspace".
- Add Managers and Team Leads with appropriate permissions.

#### Task 9: Publish the Dashboard as a Power BI App

- Customize the app name, logo, and permissions.
- Share the app with customer support managers and team leads.

#### Task 10: Embed Report in SharePoint

- Generate SharePoint Embed Link from Power BI Service.
- Add report to SharePoint Online using Power BI Web Part.

### Task 11: Enable Data Refresh in Power BI Service

- Configure scheduled refresh for live ticket updates.

### Task 12: Test User Access & Security

- Verify that:
  - Managers see all teams' data.
  - Team Leads see only their assigned teams' data.
  - Support Agents cannot access restricted data.

### Task 13: Generate & Share Report Link

- Share the Power BI App link with stakeholders.
- Create a summary PDF report for executives.

#### **Expected Outcome:**

- ✓ A dynamic & secure Customer Support Dashboard is available in Power BI Service & SharePoint.
- ✓ Managers & Team Leads can track support performance in real-time.
- ✓ Row-Level Security (RLS) ensures data confidentiality.
- ✓ The report auto-refreshes with the latest ticket data.

## **Real-Life Mini Project 1: Sales Performance Monitoring Dashboard**

#### **Objective:**

A retail company wants to track sales performance across multiple regions using Power BI Service. The dashboard should be shared securely with Regional Managers and Store Managers via Power BI Apps with Row-Level Security (RLS) applied.

**Key Features:**

- Publish report to Power BI Service for live access.
- Interactive dashboard with tiles showing sales trends, top-selling products, and region-wise performance.
- Set up RLS so that Regional Managers see only their assigned region's data.
- Embed the report in SharePoint for easy access by top management.

**Real-Life Mini Project 2: HR Employee Attendance & Productivity Report**

**Objective:**

The HR department needs a real-time attendance and productivity report to monitor employees' work hours, absenteeism, and efficiency across departments.

**Key Features:**

- Create & publish the report to Power BI Service.
- Build an interactive dashboard with drill-through to view attendance by department.
- Enable Power BI Workspaces for HR teams to collaborate on insights.
- Apply Row-Level Security (RLS) to restrict department heads from seeing other departments' data.
- Embed the report in SharePoint for HR executives and managers to access.

# Day 93

## Power BI Performance Optimization

Power BI performance optimization ensures that reports and dashboards load faster, consume less memory, and provide smooth user experience. Optimizing Power BI involves techniques such as reducing dataset size, improving DAX calculations, using aggregations, optimizing queries, and implementing incremental refresh.

### 1. Reducing Dataset Size & Optimizing DAX Calculations

#### What is it?

- Large datasets slow down report performance and increase processing time.
- Efficient DAX calculations improve speed by reducing unnecessary computations.

#### Techniques

- Remove unnecessary columns & rows before loading data.
- Use numeric keys instead of text for relationships.
- Replace calculated columns with measures where possible.

#### Syntax Example: Optimized DAX Calculation

Instead of using a Calculated Column (which takes up storage):

TotalSales\_Column = Sales[Quantity] \* Sales[Unit Price]

→ **Use a Measure instead** (which calculates dynamically without extra storage):

TotalSales = SUMX(Sales, Sales[Quantity] \* Sales[Unit Price])

## Real-Life Example: Optimizing a Sales Report

**Problem:** A sales dashboard with 100,000+ rows loads slowly due to multiple calculated columns.

### Solution:

- ✓ Remove unused columns in Power Query.
- ✓ Replace calculated columns with measures.
- ✓ Aggregate data at a higher level before loading.

### Steps to Implement:

1. Open Power Query Editor.
2. Remove unwanted columns (Right-click → Remove Columns).
3. Use Group By in Power Query to aggregate data before loading.
4. Replace calculated columns with DAX measures.

## 2. Using Aggregations & Summarization

### What is it?

- Aggregation pre-calculates summary data to improve performance.
- Helps when working with large datasets (millions of rows).

### Techniques

- Create summary tables at different granularity levels.
- Use SUMMARIZE() in DAX instead of detailed-level data queries.

### Syntax Example: Creating an Aggregation Table

```
AggregatedSales = SUMMARIZE(
    Sales,
    Sales[Region],
    Sales[ProductCategory],
    "Total Sales", SUM(Sales[SalesAmount])
)
```

This pre-aggregates sales by Region and Product Category, reducing query time.

## Real-Life Example: Monthly Sales Aggregation

**Problem:** A global sales report loads slowly due to querying millions of records.

**Solution:**

- ✓ Aggregate data at a monthly level instead of daily transactions.
- ✓ Use aggregated tables instead of raw transaction data.

**Steps to Implement:**

1. Open Power Query Editor.
2. Select Group By → Summarize by Month.
3. Load only aggregated data to Power BI.
4. Use SUMMARIZE() in DAX for high-level aggregations.

## 3. Optimizing Power Query Transformations

**What is it?**

- Power Query loads, cleans, and transforms data before it reaches the model.
- Bad transformations slow down query execution.

**Techniques**

- Disable "Auto Date/Time" in Power BI settings (reduces model size).
- Filter data early in Power Query instead of Power BI visuals.
- Use native database queries instead of slow M-language transformations.

### Syntax Example: Filtering Data in Power Query

Instead of loading all years of data, filter only recent years:

```
= Table.SelectRows(Sales, each [Year] >= 2020)
```

→ This reduces dataset size before it reaches Power BI.

## Real-Life Example: Filtering Data for Performance

**Problem:** A financial report loads data from 10 years, making it slow.

**Solution:**

- ✓ Filter out unnecessary years in Power Query.
- ✓ Remove columns that are not used in reports.

**Steps to Implement:**

1. Open Power Query Editor.
2. Select the Year Column → Click Filter Rows → Select  $\geq 2020$ .
3. Click Close & Apply.

## 4. Understanding Power BI Dataflows & Incremental Refresh

**What is it?**

- Dataflows allow you to store pre-processed data in the cloud.
- Incremental Refresh loads only new or updated data, instead of reloading the entire dataset.

**Techniques**

- Use Dataflows for large datasets instead of reloading raw data every time.
- Enable Incremental Refresh in Power BI settings.

**Syntax Example: Enabling Incremental Refresh**

1. Go to Power Query Editor.
2. Select the Date column and click "Set as Range".
3. Define Refresh Policy in Power BI:
  - a. Store data for 2 years.
  - b. Refresh only the last 1 month.

## Real-Life Example: Optimizing a Stock Market Report

**Problem:** A stock market report reloads millions of daily transactions, slowing performance.

**Solution:**

- ✓ Use incremental refresh to update only the latest transactions.
- ✓ Store only the last 1 year of data to reduce size.

**Steps to Implement:**

1. Open Power Query Editor.
2. Select the Date column → Click "Set as Parameter".
3. Enable Incremental Refresh in Power BI settings.

### Summary of Key Optimizations

Optimization	Techniques Used
Reducing Dataset Size	Remove unnecessary columns, use Measures instead of Calculated Columns
Using Aggregations	Precompute summaries using SUMMARIZE()
Optimizing Power Query	Filter data early, disable Auto Date/Time
Using Dataflows & Incremental Refresh	Store preprocessed data, load only new data

## Mini Project 1: Optimizing a Sales Dashboard for an E-commerce Business

**Project Objective:**

An e-commerce company has a sales dashboard in Power BI that takes too long to load. The goal is to optimize the dataset, improve DAX calculations, and use aggregations to make the dashboard run faster.

**Dataset:**

- Sales Transactions (SalesTable.csv)
- Customer Details (Customers.csv)
- Product List (Products.csv)

**Step-by-Step Implementation:****Step 1: Load the Dataset in Power BI**

1. Open Power BI Desktop.
2. Click Home → Get Data → CSV.
3. Load the datasets (SalesTable.csv, Customers.csv, Products.csv).
4. Click Transform Data to enter Power Query Editor.

**Step 2: Reduce Dataset Size & Optimize Data Model****Remove Unused Columns:**

- In Power Query, delete unnecessary columns (e.g., Order Notes, Customer Address).

**Filter Data to Include Only Last 2 Years:**

```
= Table.SelectRows(SalesTable, each [OrderDate] >=
Date.AddYears(DateTime.LocalNow(), -2))
```

**Change Data Types Efficiently:**

- Convert text columns (like Order ID) to **integer format** where possible.
- Convert Sales Amount to **Decimal Number**.

**Result:** This reduces the dataset size and speeds up calculations.

**Step 3: Create Efficient DAX Measures Instead of Calculated Columns**

**✗ Bad Practice:** Using a calculated column to calculate total sales.

```
TotalSales_Column = SalesTable[Quantity] * SalesTable[Unit Price]
```

**X Problem:** Calculated columns take extra memory and slow down performance.

**Best Practice:** Use a DAX measure instead:

```
TotalSales = SUMX(SalesTable, SalesTable[Quantity] * SalesTable[Unit Price])
```

**Result:** This calculates dynamically without storing extra data.

#### **Step 4: Use Aggregations to Improve Query Speed**

**Create an Aggregated Sales Table:**

```
SalesSummary = SUMMARIZE(
    SalesTable,
    SalesTable[ProductCategory],
    SalesTable[OrderYear],
    "Total Sales", SUM(SalesTable[SalesAmount])
)
```

Use This Table Instead of Raw Data in Reports.

**Result:** Queries run much faster as Power BI only reads pre-aggregated data.

#### **Step 5: Enable Incremental Refresh to Reduce Data Load**

**Enable Incremental Refresh in Power BI:**

1. In Power Query, set OrderDate as a Date/Time parameter.
2. Click File → Options → Incremental Refresh.
3. Set "Store data for 2 years" and "Refresh only the last 1 month".

**Result:** Instead of reloading the full dataset daily, Power BI only loads new transactions.

### **Step 6: Optimize Report Performance**

#### **Disable Auto Date/Time in Power BI:**

1. Go to File → Options → Global → Data Load.
2. Uncheck "Auto Date/Time".

**Result:** Prevents Power BI from creating unnecessary hidden date tables.

#### **Use Composite Models (Import + DirectQuery) for Large Datasets**

1. Keep recent sales data in Import Mode.
2. Use DirectQuery for older data stored in a SQL database.

**Result:** Reduces RAM usage and improves loading speed.

#### **Final Outcome:**

- ✓ Sales Dashboard loads 5x faster
- ✓ Dataset size reduced by 50%
- ✓ Reports use optimized DAX and aggregations

### **Mini Project 2: Optimizing Financial Reporting for a Bank**

#### **Project Objective:**

A bank needs a financial dashboard in Power BI, but large transaction datasets cause slow loading. The goal is to use aggregations, reduce data size, and enable incremental refresh for faster performance.

#### **Dataset:**

- Transactions (Transactions.csv)

- Customer Accounts (Accounts.csv)
- Loan Details (Loans.csv)

## Step-by-Step Implementation:

### Step 1: Load and Clean Data in Power Query

1. Remove Unnecessary Columns (e.g., Customer Address, Branch Phone Number).

2. Filter Data to keep only transactions from the last 5 years:

```
= Table.SelectRows(Transactions, each [TransactionDate] >= Date.AddYears(DateTime.LocalNow(), -5))
```

**Result:** Dataset is smaller and more efficient.

### Step 2: Optimize DAX Calculations for KPIs

**Problem:** Financial analysts need Total Loan Amount and Total Deposits metrics.

**Solution:** Instead of Calculated Columns, create DAX Measures:

#### Measure for Total Loans:

```
TotalLoans = SUM(Loans[LoanAmount])
```

#### Measure for Total Deposits:

```
TotalDeposits = SUM(Transactions[DepositAmount])
```

**Result:** These measures calculate on demand, improving performance.

### **Step 3: Implement Aggregations for Faster Reporting**

#### **Create an Aggregated Summary Table for Monthly Transactions:**

```
MonthlySummary = SUMMARIZE(  
    Transactions,  
    Transactions[Month],  
    Transactions[Branch],  
    "Total Deposits", SUM(Transactions[DepositAmount]),  
    "Total Withdrawals", SUM(Transactions[WithdrawalAmount])  
)
```

**Result:** Power BI now queries a small summary table instead of millions of rows.

### **Step 4: Enable Incremental Refresh for Transaction Data**

1. Set Up Date Parameters in Power Query.
2. Go to File → Options → Incremental Refresh.
3. Set "Store 5 years of data" and "Refresh only the last 3 months".

**Result:** The bank dashboard loads new transactions only, instead of all data.

### **Step 5: Use DirectQuery for High-Volume Data**

**Problem:** Querying millions of transactions slows performance.

**Solution:** Use DirectQuery for historical data (older than 1 year), while keeping recent data in Import Mode.

#### **Steps:**

1. Set last 1 year of transactions in Import Mode.
2. Use DirectQuery for older transactions stored in a SQL database.

**Result:** Queries are fast and memory-efficient.

### Step 6: Optimize Data Model & Report Visuals

- ✓ Disable "Auto Date/Time" to reduce unnecessary calculations.
- ✓ Use Bookmarks & Drill-Through Pages to load only relevant visuals.
- ✓ Use Hierarchies (Year → Quarter → Month) instead of multiple filters.

**Result:** Dashboard becomes more responsive and user-friendly.

### Final Outcome:

- ✓ Financial dashboard loads 4x faster
- ✓ Only new transactions refresh, reducing data processing time
- ✓ Bank reports work efficiently with optimized DAX and DirectQuery

### Summary of Power BI Performance Optimization Techniques

Technique	Benefit
Reduce dataset size	Improves speed by filtering and removing unused data
Use DAX measures instead of calculated columns	Saves memory and enhances performance
Create aggregated tables	Reduces query time
Enable incremental refresh	Loads only new data, making refresh faster
Use DirectQuery for large datasets	Reduces RAM usage and speeds up queries

## Real-Life Mini Project: Optimizing a Retail Sales Dashboard for Power BI

### Project Objective:

A retail company needs to improve the performance of its Power BI sales dashboard. Currently, the dashboard is slow due to a large dataset containing millions of sales transactions. The goal is to reduce dataset size, optimize DAX calculations, use aggregations, and implement incremental refresh to enhance speed and efficiency.

### Dataset:

- SalesTransactions.csv – Sales data (millions of records)
- Products.csv – Product details
- Customers.csv – Customer information
- Stores.csv – Store details

### Day 93 Tasks

1. Load the dataset into Power BI and inspect the data model.
2. Remove unnecessary columns and filter data in Power Query to keep only the last 3 years of sales.
3. Convert text columns to integer where possible (e.g., Order ID, Customer ID) to reduce storage size.
4. Create aggregated sales summary tables at monthly and category levels using DAX or Power Query.
5. Replace calculated columns with DAX measures for key metrics like Total Sales, Profit Margin, and Average Order Value.
6. Optimize slow DAX calculations using SUMX, VAR, and RETURN statements instead of inefficient formulas.
7. Enable incremental refresh by setting up date-based parameters to refresh only new sales transactions instead of the entire dataset.

8. Use DirectQuery for older sales data (older than 3 years) while keeping recent sales data in Import Mode for faster analysis.
9. Optimize Power Query transformations by performing data filtering, joins, and calculations in SQL or the source system instead of Power BI.
10. Use aggregations in Power BI to pre-calculate Total Sales, Total Profit, and Units Sold per Product Category and Region.
11. Disable Auto Date/Time in Power BI settings to prevent unnecessary hidden date tables.
12. Implement hierarchy-based filtering (Year → Quarter → Month) to improve slicer performance instead of using multiple filters.
13. Monitor report performance using Power BI Performance Analyzer and identify bottlenecks in visuals, queries, and DAX calculations.

**Expected Outcomes:**

- ✓ Dashboard loads faster with optimized dataset and DAX calculations.
- ✓ Queries execute quickly using aggregations and DirectQuery for older data.
- ✓ Power BI refresh time reduces significantly due to incremental refresh.
- ✓ Smoother user experience with optimized slicers, visuals, and calculations.

This project ensures the retail company can analyze sales data efficiently while keeping Power BI performance optimal.

## Project 1: Optimizing an E-Commerce Sales Dashboard

### Objective:

An e-commerce company tracks millions of orders across various regions. The existing Power BI dashboard is slow due to large datasets and inefficient calculations. The goal is to optimize dataset size, improve DAX performance, and implement incremental refresh to enhance report speed.

### Dataset:

- Orders.csv – Order transactions (millions of rows)
- Customers.csv – Customer details
- Products.csv – Product categories and prices
- Shipping.csv – Shipping details per order

### Key Optimization Goals:

- ◆ Reduce dataset size by keeping only last 2 years of data
- ◆ Use aggregations to precompute total sales per region
- ◆ Replace calculated columns with optimized DAX measures
- ◆ Enable incremental refresh for the Orders table
- ◆ Optimize Power Query transformations to avoid performance issues

## Project 2: Enhancing Performance of a Financial Report in Power BI

### Objective:

A financial institution uses Power BI to track revenue, expenses, and profitability. However, their dashboard is slow due to large financial transactions and complex DAX measures. This project focuses on DAX optimization, dataflows, and summarization techniques.

**Dataset:**

- FinancialTransactions.csv – Detailed revenue and expenses (millions of rows)
- CostCenters.csv – Cost allocation per department
- ProfitabilityMetrics.csv – Key performance indicators

**Key Optimization Goals:**

- ◆ Reduce dataset size by summarizing financial transactions at a monthly level
- ◆ Use Power BI Dataflows to precompute data before loading into Power BI
- ◆ Replace inefficient CALCULATE() and FILTER() DAX formulas with optimized versions
- ◆ Implement incremental refresh for financial data updates
- ◆ Improve visualization performance by reducing the number of high-cardinality fields

## Day 94

### Final Project – Creating a Full Power BI Dashboard

#### Introduction

A Power BI dashboard is an interactive report that combines data from multiple sources, processes it, and presents key insights using visualizations, tables, and metrics. This project will walk through the full implementation process, covering data import, transformation, modeling, DAX calculations, advanced visualizations, and publishing.

## Project Overview – Sales Performance Dashboard

### Objective

A retail company wants to track sales performance across various regions, products, and time periods. The goal is to create a dynamic Power BI dashboard that allows business users to:

- ✓ Analyze total revenue, profit, and sales trends
- ✓ Compare performance by region and product category
- ✓ Identify top-selling products and least profitable regions
- ✓ Implement dynamic filtering and drill-downs

### Step 1: Data Import & Transformation (Power Query)

#### Dataset:

We will work with three datasets:

1. Orders.csv – Contains sales transactions
2. Products.csv – Includes product details (category, price, etc.)
3. Customers.csv – Stores customer demographics

#### Steps to Import Data into Power BI

1. Open Power BI Desktop and click on "Get Data"
2. Select CSV (for local files) or SQL Server, Excel, APIs (for other sources)
3. Load the three datasets into Power Query Editor

#### Data Cleaning & Transformation

- Remove null values in columns like Order Date, Customer ID
- Change data types (e.g., convert Order Date to Date format)
- Remove duplicates in Customers and Products tables
- Merge tables to bring relevant fields together
- Unpivot data if needed for better analysis

## Step 2: Data Modeling & Relationships

### Creating Relationships

- Connect Orders → Products (using Product ID)
- Connect Orders → Customers (using Customer ID)

### Understanding Schema

- We will use a Star Schema where the Orders table is the Fact Table, and Customers & Products are Dimension Tables.
- Ensure One-to-Many relationships between Fact & Dimension tables

## Step 3: DAX Calculations (Data Analysis Expressions)

### Creating Key Performance Metrics

#### 1. Total Sales

Total Sales = SUM(Orders[Sales])

#### 2. Total Profit

Total Profit = SUM(Orders[Profit])

#### 3. Average Order Value

Avg Order Value = DIVIDE([Total Sales], DISTINCTCOUNT(Orders[Order ID]))

#### 4. Top-Selling Products (Ranked)

Top Products = RANKX(ALL(Products), [Total Sales], , DESC, DENSE)

#### 5. Sales Growth Over Previous Year

Sales Growth = CALCULATE([Total Sales], SAMEPERIODLASTYEAR(Orders[Order Date]))

## Step 4: Advanced Visualizations

### Adding Interactive Charts

- ◆ Sales Trend (Line Chart) – X-Axis: Order Date, Y-Axis: Total Sales
- ◆ Top 5 Products (Bar Chart) – X-Axis: Product Name, Y-Axis: Total Sales
- ◆ Sales by Region (Map Visualization) – Location: Region, Size: Sales
- ◆ Profit Margin (Card KPI) – Total Profit / Total Sales

### Enhancing Visuals

- ✓ Conditional Formatting – Highlight low-profit regions in red
- ✓ Drill-through – Click on a region to see product-wise sales
- ✓ Hierarchies – Add Year > Quarter > Month in date filters
- ✓ Slicers – Allow users to filter by Region, Product, Year

## Step 5: Publishing & Sharing Dashboards

1. Save the Power BI report (.pbix file)
2. Click "Publish" → Upload to Power BI Service
3. Create a Dashboard with Key Tiles
4. Share with specific users using row-level security (RLS)
5. Embed the dashboard into SharePoint or a Website

### Final Outcome

- A fully interactive Power BI dashboard displaying sales insights
- Optimized data model ensuring fast performance
- Advanced DAX calculations for deeper analysis
- Dynamic filtering & drill-downs for user-friendly experience

## Business Impact

- ✓ Faster decision-making with interactive dashboards
- ✓ Identifies trends & opportunities using data-driven insights
- ✓ Enhances collaboration by sharing reports across teams

## 1. Mini Project: Customer Satisfaction Analysis Dashboard

### Project Overview

A service-based company wants to analyze customer feedback to improve its service quality. The goal is to create a Customer Satisfaction Dashboard that provides insights into customer ratings, feedback trends, and areas of improvement.

### Key Requirements

- Data Import & Transformation: Load customer feedback data from multiple sources (Excel, SQL, Online Surveys).
- Data Modeling & Relationships: Connect survey responses with customer demographics and service details.
- DAX Calculations: Calculate average customer rating, Net Promoter Score (NPS), percentage of positive & negative feedback.
- Advanced Visualizations: Use bar charts for rating distribution, word cloud for text analysis, and trend analysis for monthly satisfaction scores.
- Publishing & Sharing: Publish to Power BI Service and set up row-level security (RLS) to allow managers to see only their region's data.

## 2. Mini Project: Employee Performance & Productivity Dashboard

### Project Overview

A company wants to track employee productivity and performance across various departments. The goal is to create an Employee Performance Dashboard to monitor key metrics such as attendance, project completion rate, and employee engagement.

### Key Requirements

- Data Import & Transformation: Import HR data, attendance records, and project completion reports from different sources.
- Data Modeling & Relationships: Establish relationships between employee records, departments, and performance metrics.
- DAX Calculations: Calculate employee efficiency score, average working hours, task completion rate, and attrition trends.
- Advanced Visualizations: Use heatmaps for attendance trends, gauge charts for productivity scores, and department-wise performance comparisons.
- Publishing & Sharing: Publish to Power BI Service and implement role-based access so that managers can view only their team's performance.