Lessons Book - Python Programming

Inspire Academy, August 2025

Contents

Lesson 01 **10**
II. Setting up Python Environment . . . 10
Iii. Python Basics- Language, History, Pep Standards, And Indentation . . . 12
III. Python Basics: Language, History, PEP Standards, and Indentation . . . 12
Python's Global Community . . . 16
Iv. Python Syntax, Variables, And Data Types . . . 16
IV. Python Syntax, Variables, and Data Types . . . 16
Lesson 1- Introduction To Python Basics . . . 20

LESSON 1: INTRODUCTION TO PYTHON BASICS **20**
Learning Objectives . . . 20
1. Setting Up the Python Environment (15 minutes) . . . 20
2. Introduction to Python . . . 20
Readme . . . 21

Python Programming **21**
V. Input:Output Functions, Print Statements, And Console Communication . . . 21
V. Input/Output Functions, Print Statements, and Console Communication (20 minutes) 21
Vi. Hands-On Coding- Simple Python Programs . . . 24
VI. Hands-on Coding: Simple Python Programs (20 minutes - work) . . . 24
Vii. Q&A And Wrap-Up . . . 27
VII. Q&A and Wrap-Up (10 minutes) . . . 27

Lesson 02 **29**
Ii. Operators, Counting, Flow Control, Enumerators . . . 29
II. Operators, Counting, Flow Control, Enumerators . . . 29
Summary Table . . . 33
Let us check play with: . . . 33
Iii. Conditional Statements . . . 33
III. Conditional Statements . . . 33
Iv. Loops- For And While, Break And Continue . . . 39
IV. Loops: for and while, break and continue . . . 39
V. Combined Hands-On Assignment- Python Control Flow Mini-Project . . . 45
V. Combined Hands-On Assignment: Python Control Flow Mini-Project . . . 45
Sample Program Structure . . . 46
Instructor Notes & Learning Objectives . . . 47

Extension Ideas (for advanced students or extra credit): . . . 47
Vi.Wrap Up And Homework . . . 47
VI. Wrap Up & Homework Assignment . . . 47

Lesson 03 **49**
Exercise . . . 49
LESSON 3: PYTHON CONTROL FLOW (Exercises) . . . 49
Lesson 3- Python Control Flow (Solutions) . . . 52

LESSON 3: PYTHON CONTROL FLOW (Exercises & Solutions) **52**
Instructions . . . 52

Lesson 04 **57**
Ii. Introduction To Lists And Tuples . . . 57
LESSON 4: PYTHON DATA STRUCTURES - PART 1 (LISTS AND TUPLES) 57
II. Introduction to Lists and Tuples (15–20 min) . . . 57
Iii. List Operations . . . 60
III. List Operations . . . 60
Iv. Tuple Operations . . . 64
IV. Tuple Operators and Operations . . . 64
V. Hands-On Practice . . . 67
V. Hands-On Practice (20 min) . . . 67
Vi. Homework Assignment . . . 70
VI. Homework Assignment . . . 70

Lesson 05 **71**
Python Data Structures - Part 1-Lab . . . 71
Lesson 5: Python Data Structures - Part 1 . . . 71
Solution-Python Data Structures - Part 1-Lab . . . 73
Solutions: Python Data Structures - Tuples and Lists . . . 73

Lesson 06 **76**
Ii. Introduction To Dictionaries And Sets . . . 76
LESSON 6: PYTHON DATA STRUCTURES - PART 2 (DICTIONARIES AND SETS) . . . 76
II. Introduction to Dictionaries and Sets . . . 76
Iii.Dictionary Operations . . . 78
Iv. Introduction To Sets . . . 81
V. Hands-On Practice . . . 84
V. Hands-on Practice . . . 84
VI. Homework . . . 85

Lesson 07 **87**
Lesson 7- Python Data Structures - Part 2 - Lab Solutions . . . 87
LESSON 7: PYTHON DATA STRUCTURES - PART 2 (Exercise) - Solution . . . 87
Lesson 7- Python Data Structures - Part 2 - Lab . . . 91
LESSON 7: PYTHON DATA STRUCTURES - PART 2 (Exercise) . . . 91

Lesson 08 **93**

I. Hands-On Coding- Breaking Down Problems With Functions . . . 93
VI. Hands-on Coding: Breaking Down Problems with Functions . . . 93
Ii. Defining Functions In Python . . . 95
II. Defining Functions in Python . . . 95
Iii. Function Arguments, Return Values, And Scope . . . 98
III. Function Arguments, Return Values, and Scope . . . 98
Iiv. Homework . . . 102
VIII. Homework Assignment . . . 102
Iv. Advanced Functions- Decorators, Anonymous, And Lambda Functions . . . 104
IV. Advanced Functions: Decorators, Anonymous, and Lambda Functions . . . 104
Outline . . . 106
Lesson 8: Python Functions . . . 106

Lesson 09 **107**
Exercise . . . 107
LESSON 09: Additional Python Function exercises (with solutions) . . . 107
Solutions . . . 108
Exercise . . . 110
Lesson 09: Python Functions (Exercise) . . . 110
Lesson 09- Python Functions (Solutions) . . . 113
Lesson 09: Python Functions (Solutions) . . . 113

Lesson 10 **117**
I. Introduction To Pandas . . . 117
I. Introduction to Pandas . . . 117
Ii. Loading Data Into Pandas . . . 122
II. Loading Data into Pandas . . . 122
Iii. Dataframes And Basic Operations . . . 127
III. DataFrames and Basic Operations . . . 127
Iv. Hands-On Practice With Pandas . . . 131
IV. Hands-on Practice with Pandas . . . 131
V.Homework . . . 134
V. Homework Assignment: Exploring a CSV Dataset with Pandas . . . 134
Solution Code (with steps) . . . 134

Lesson 11 **135**
Exercise . . . 135
LESSON 11: MANIPULATING DATA IN PYTHON – PART 1 . . . 135
Dataset: `employee_performance.csv` . . . 135
Additional: . . . 139
Lesson 11- Python Data Manipulationi (Solutions) . . . 139
LESSON 11: SOLUTIONS . . . 139
Part-G. Polars Solutions (Assignments 16–20) . . . 141

Lesson 12 **142**
I.Data Cleaning With Pandas . . . 142
I. Data Cleaning with Pandas . . . 142
Ii. Data Aggregation In Pandas . . . 146

II. Data Aggregation in Pandas . . . 146
Iii. Working With Date And Time In Pandas . . . 149
III. Working with Date and Time in Pandas . . . 149
Iv. Working With Strings In Pandas . . . 150
IV. Working with Strings in Pandas . . . 150
V. Working With Index And Multiindex . . . 153
V. Working with Index and MultiIndex . . . 153
Vi. Working With Data Types And Basic Windowing Function . . . 154
VI. Working with Data Types and basic windowing function . . . 154
Basic Windowing Functions . . . 155
Vii.Hands-On Practice . . . 155
VII. Hands-on Practice with `employee_performance.csv` . . . 155
Outline . . . 158

**** LESSON 12: MANIPULATING DATA IN PYTHON – PART 2**** **158**
DATA AGGREGATION AND CLEANING . . . 158
Recap of Lesson 11 . . . 158
1. Data Cleaning in Pandas . . . 158
2. Data Aggregation in Pandas . . . 159
3. Hands-on Practice . . . 160
4. Homework Assignment . . . 160
Additional Topics to Cover in Lesson 12 . . . 160

Lesson 13 **163**
Exercise . . . 163
LESSON 13: Exercise . . . 163
Dataset: `sales_data.csv` . . . 163
Assignments . . . 164
Lesson 13- Solutions . . . 164
LESSON 13: Solutions . . . 164

Lesson 14 **167**
0.Concepts . . . 167
Concepts . . . 167
I.Foundations Of Relational Databases . . . 169
Ii. Sql Basics For Python Programmers . . . 172
Iii. Sql Joins And Relationships . . . 175
Iv. Hans-On Practice . . . 178
V.Homework . . . 181
LESSON 14: INTRODUCTION TO SQL FOR PYTHON DEVELOPERS 181

Lesson 15 **182**
Ii. Advanced Sql Techniques . . . 182
II. Advanced SQL Techniques . . . 182
Summary: JOIN Selection Guide . . . 190
Iii. Query Optimization For Python Applications . . . 190
III. Query Optimization for Python Applications . . . 190
Exercise . . . 195

IV. Hands-on Exercise: Advanced SQL Techniques & Index Optimization . . . 195
Outline . . . 198
LESSON 15: ADVANCED SQL CONCEPTS WITH PYTHON APPLICATIONS . . . 198

Lesson 16 **200**
Ii. Introduction And Database Setup . . . 200

LESSON 16: INTEGRATING SQL WITH PYTHON **200**
II. Introduction and Database Setup . . . 200
Topics Covered: . . . 200
Key Concept . . . 200
Tool of Choice: `sqlite3` . . . 200
1. Creating a SQLite Database and Table . . . 200
How SQLite Creates a Database . . . 200
How to create a Table if Not Exists . . . 201
2. Repeated Connections — Safe Way with Context Manager . . . 202
3. Reading Data from the Table . . . 202
Summary of Basic Database Operations . . . 202
Important: Simple Error Handling Example . . . 203
Recap: . . . 203
Iii. Performing Sql Operations In Python . . . 203
III. Performing SQL Operations in Python . . . 203
Topics Covered: . . . 203
Key Concept . . . 203
Basic SQL Operations in Python . . . 204
Summary of SQL Operations with Examples . . . 205
Why Use Placeholders (`?`) . . . 206
Practical Example — Combined Operations . . . 206
Recap: . . . 206
Iv. Advanced Queries And Integration With Functions Pandas . . . 206

IV. Advanced Queries and Integration with Functions & Pandas **206**
Topics Covered: . . . 207
Key Concept . . . 207
1. Setting Up Related Tables . . . 207
2. Insert Sample Data for JOIN Demonstration . . . 207
3. Performing INNER JOIN with Python & SQLite . . . 208
4. Integrating with Pandas for Data Analysis . . . 208
5. (optional): Use Pandas to Explore Data . . . 209
Concepts . . . 209
Recap . . . 209
Sqlalchemy_Additional . . . 209
SQLAlchemy . . . 209
Sqlalchemy_Short . . . 211
SQLAlchemy . . . 211
ORM like / without typical SQL . . . 212
V. Hands-On Practice And Homework Assignment . . . 213

V. Hands-On Practice and Homework Assignment **213**
Purpose of this Section . . . 213
Key Concept: Transactions & Fetching Results . . . 213
Hands-On Demo: Transactional Update and Fetch . . . 214
Homework Assignment: Build a Complete Python Mini-App . . . 215
Example Homework Structure Suggestion . . . 216
Outline . . . 216
LESSON 16: INTEGRATING SQL WITH PYTHON . . . 216
Outline2 . . . 219

LESSON 16: INTEGRATING SQL WITH PYTHON **219**
SECTION 1: Introduction and Database Setup . . . 219
SECTION 2: Performing SQL Operations in Python . . . 220
SECTION 3: Advanced Queries and Integration with Functions & Pandas . . . 221
SECTION 4: Hands-On Practice + Homework Assignment . . . 222

Lesson 17 **223**
Exercise . . . 223
LESSON 17: INTEGRATING SQL WITH PYTHON (Exercise) . . . 223
Lesson 17- Integrating Sql With Python (Solution) . . . 224

LESSON 17: INTEGRATING SQL WITH PYTHON (Solution) **224**
Part 1: Core SQL Assignments (1-10) . . . 224
Part 2: SQL with Python (sqlite3) Assignments (11-20) . . . 226

Lesson 18 **228**
Ii. Creating Layers And Adding Geometrics To Data . . . 228
II. Creating Layers and Adding Geometrics to Data . . . 228
Iii. Understanding Types Of Graphs For Types Of Variables . . . 230
III. Understanding Types of Graphs for Types of Variables . . . 230
Iv. Adding Text, Grid, Lines, And Legends To Graphs . . . 234
IV. Adding Text, Grid, Lines, and Legends to Graphs . . . 234
Lesson18 . . . 237

Introduction to Matplotlib in Python **237**
What is Matplotlib? . . . 237
Installation . . . 237
Importing Matplotlib . . . 237
Sample Dataset . . . 237
Basic Plot Types . . . 238
Customization Options . . . 239
Subplots . . . 239
Styling with `plt.style` . . . 239
Advanced Features . . . 240
Saving Plots . . . 240
Interactive Plots (Optional with `%matplotlib notebook` or `matplotlib.widgets`) . . . 240
Resetting the Plot . . . 241
Summary of Key Components . . . 241

V. Extended Dataset With Date And Extra Models . . . 241
V. Extended Dataset with Date and Extra Models . . . 241
Visual Examples Covering All Graph Types & Layers . . . 242
Reusable Plotting Function Example . . . 245
Summary of What's Covered . . . 247
Or . . . 247
Summary of Graph Types & Use-Cases . . . 247
Conclusion . . . 247
Vi. Hands-On + Homework . . . 247
VI. Hands-on: Working with Data to Prepare 3 Different Plots . . . 247
VII. Homework Assignment (Optional or To Continue at Home) . . . 249
Bonus Challenge: . . . 250

Lesson 19 **250**
Ensuring Plots Display Well In This Environment . . . 250
Exercise . . . 253
Exercise . . . 257

Lesson 20 **259**
1. Handling Exceptions In Python . . . 259
Common Python Errors & Exceptions . . . 259
Best Practices for Handling Exceptions . . . 263
Ii.Errors And Exceptions . . . 264
II. Errors and exceptions . . . 264
Sample Dataset (for demo programs) . . . 264
1. How `try…except…else…finally` Works . . . 264
2. Demo Programs Covering Exception Types . . . 265
3. Summary Table . . . 266
4. Recap of Covered Built-in Exceptions . . . 267
Putting It All Together: Full Script Example . . . 267
Iii. Exceptions . . . 268
III. Exceptions . . . 268
Demo 1: Catching Multiple Exception Types . . . 268
Demo 2: Catching Unknown Exceptions . . . 269
Demo 3: Using `else` and `finally` . . . 269
Demo 4: Custom Exception . . . 269
Iv. Handling And Raising Exceptions . . . 270
IV. Handling and raising exceptions . . . 270
Basic Structure . . . 270
Demo 1: Handling a Specific Exception . . . 270
Demo 2: Handling Multiple Exceptions in One Line . . . 271
Demo 3: Using `else` and `finally` . . . 271
Demo 4: Handling Unknown Exceptions . . . 271
Re-raising Exceptions . . . 271
Best Practices . . . 272
Challenge Demo: Full Exception Handling Flow . . . 272
Keyboardinterrupt . . . 272
V. Clean-Up Actions . . . 274

V. Clean-up actions . . . 274
Use Case: Resource Management . . . 274
Methods for Clean-up: . . . 274
Demo 3: `finally` Always Executes . . . 275
Summary Table . . . 275
Demo 4: Cleaning Up a Network Connection (Simulated) . . . 276
Best Practices . . . 276
Vi. Modules . . . 276
VI. Modules . . . 276
1. **Creating a Module** . . . 277
2. **Importing Modules** . . . 277
3. **Module Search Path** . . . 277
4. **The `__name__ == "__main__"` Trick** . . . 278
5. **Standard Modules** . . . 278
6. **Packages (Brief Overview)** . . . 278
Practical Demo: Using Standard Modules . . . 279
Summary Table . . . 279
Best Practices . . . 279
Python Modules and Packages, Requisites, and Environments . . . 280
1. What Are Python Modules and Packages? . . . 280
2. What Are Requisites? . . . 280
Example: `requirements.txt` . . . 280
Code Using Requisites (`http_demo.py`) . . . 280
How to Generate `requirements.txt` . . . 281
Example: Reading and Installing . . . 281
Virtual Environments (Isolated Python Spaces) . . . 281
Recap Summary . . . 282
Best Practices . . . 282
Vii. Hands-On . . . 282
VI. Hands-on . . . 282
Content . . . 284
LESSON 20: ERROR HANDLING AND MODULES . . . 284

Lesson 21 **286**
Lesson 21 Error Handling And Modules – Solutions . . . 286
Lesson 21: Error Handling and Modules – Exercises . . . 286
Solutions . . . 286
Exercise . . . 290
Lesson 21: Error Handling and Modules – Exercises . . . 290
Error_Handling . . . 292

Part 1 **292**

Part 2 **295**
Assignment 1: Using the `math` Module . . . 295
Assignment 2: Working with `random` Module . . . 295
Assignment 3: Using `datetime` Module . . . 296
Assignment 4: Get Module Information with `dir()` and `help()` . . . 296

Assignment 5: Use the `os` Module to List Files . . . 296
Assignment 6: Checking Module Versions . . . 297
Assignment 7: Import Functions Directly from a Module . . . 297
Assignment 8: Creating a Custom Module . . . 297
Assignment 9: Module with Multiple Functions . . . 297
Assignment 10: `if __name__ == "__main__"` in a Module . . . 298

Lesson 22 **298**
I. What Is Oop . . . 298

I. What is OOP in Python **298**
1. Object-Oriented Programming (OOP) . . . 298
2. Names and Objects in Python . . . 300
3. Scopes and Namespaces . . . 300
4. Three New Object Types Introduced with Classes . . . 301
5. Putting It All Together . . . 302
Ii. Key Oop Concepts In Python . . . 303
II. Key OOP Concepts in Python . . . 303
Exercise . . . 307
6. Practical Examples . . . 307
Iii. Hands-On Example- Modeling Real-World Objects (Atm) . . . 313
III. Hands-on Example: My First ATM Script :) . . . 313
4. Summary Table (Aligned to Bank Story) . . . 316

Lesson 23 **316**
Atm Project Overview . . . 316
ATM Project Overview . . . 316
Project Features . . . 317
Atm Project – Full Code . . . 320
ATM Project – Full Code Skeleton . . . 320
I.Short Review . . . 326
SECTION 1: General Python Understanding . . . 326
SECTION 2: Data Wrangling . . . 327
SECTION 3: Data Visualization . . . 328
SECTION 4: Modules, Classes & OOP . . . 328

Lesson 24 **329**
Mini Project Assignement . . . 329

Car Rental System – Mini assignment for certificate **329**
1. Goals of the Assignment . . . 329
2. Core Functionalities . . . 329
3. Database Schema (SQLite) . . . 331
4. File Logging . . . 331
5. Input Validation & Error Handling . . . 331
6. OOP Structure . . . 331
7. Data Wrangling & Visualization . . . 332
8. **Submission Instructions** . . . 332

Lesson 01

II. Setting up Python Environment

A. What is an IDE?

- **IDE (Integrated Development Environment):** A software application for writing, editing, and running code. Popular for Python: PyCharm, VS Code, Thonny.
- **Jupyter Notebooks:** Interactive coding environment, great for data science, visualization, and prototyping.

B. Step 1: Install Python (Required for all IDEs)

1. **Go to python.org/downloads.**
2. **Download the latest version** (recommended: Python 3.x).
3. **Run the installer:**
   - On Windows: Select "Add Python to PATH" **before** clicking "Install Now."
   - On macOS: Drag the Python icon to Applications.
   - On Linux: Most distributions have Python pre-installed; else, use package manager.
4. **Verify Installation:**
   - Open Terminal (macOS/Linux) or Command Prompt (Windows).
   - Type:

     ```
     python --version
     ```

     or

     ```
     python3 --version
     ```

C. Step 2: Install IDEs

1. PyCharm (Community Edition - Free)

- **Download:** jetbrains.com/pycharm/download
- **Install:**
  1. Choose Community Edition (free).
  2. Run installer (Windows/macOS/Linux).
  3. Launch PyCharm.
  4. On first launch, select "New Project," set interpreter to the Python installed earlier.
- **Screenshot suggestion:** Show the PyCharm welcome screen.

2. Visual Studio Code (VS Code)

- **Download:** code.visualstudio.com
- **Install:**
  1. Run the installer.
  2. Launch VS Code.
  3. Go to Extensions (left bar), search for "Python," and install the official Python extension (by Microsoft).
  4. Restart VS Code if prompted.
- **Set Interpreter:** Press `Ctrl+Shift+P`, type "Python: Select Interpreter," and choose your Python 3.x install.
- **Screenshot suggestion:** Show VS Code with Python extension installed.

3. Thonny (Beginner-friendly)

- **Download:** thonny.org
- **Install:**
  1. Download the installer for your OS.
  2. Run the installer and follow prompts.
  3. Thonny usually comes with Python bundled; no need for extra setup.
  4. Open Thonny and you're ready to code!
- **Screenshot suggestion:** Show Thonny's simple interface.

D. Step 3: Install Jupyter Notebooks

Option 1: Via Anaconda (Easiest for beginners/data science)

- **Download Anaconda:** anaconda.com/products/distribution
- **Install Anaconda.** (Follows OS-specific prompts.)
- **Launch "Anaconda Navigator"** and click "Launch" under Jupyter Notebook.

Option 2: Using pip (for those comfortable with command line)

- Open Terminal/Command Prompt.
- Run:

  ```
  pip install notebook
  ```

- Then start Jupyter:

  ```
  jupyter notebook
  ```

- Browser will open at `localhost:8888` showing the Jupyter dashboard.

Screenshot suggestion: Show the Jupyter Notebook dashboard in the browser.

E. Optional: Online Interpreters

- repl.it/languages/python3
- Google Colab
- Use these if unable to install software locally.

F. Demo Activity: "Hello, World!" in Each Environment

Instructions for Students:

- Open your chosen environment.
- Type and run the following program:

```
print("Hello, World!")
```

How to Run:

- **PyCharm:**
  - Create New Project > New Python File > Type code > Right-click > Run.
- **VS Code:**
  - Open folder > New File (`hello.py`) > Type code > Right-click > "Run Python File in Terminal."
- **Thonny:**
  - Type code in editor > Click the green "Run" button.
- **Jupyter Notebook:**
  - Click "New Notebook" > Type code in a cell > Press Shift+Enter.

Expected Output:

```
Hello, World!
```

G. Quick Troubleshooting

- If Python not recognized: Close and reopen terminal or restart your computer.
- Make sure the interpreter is set to Python 3.x in your IDE.
- Use print statements to check your environment.

Iii. Python Basics- Language, History, Pep Standards, And Indentation

III. Python Basics: Language, History, PEP Standards, and Indentation

---

1. Python: A Quick Overview

- **Python** is a high-level, interpreted programming language.
- Known for **readability**, **simplicity**, and a huge supportive community.
- Widely used for web development, automation, data science, AI, education, scripting, and more.

---

2. A Brief History of Python

- **Creator:** Guido van Rossum, Netherlands.
- **First released:** 1991 (Python 0.9.0).
- **Name origin:** Named after the TV show "Monty Python's Flying Circus," not the snake!
- **Philosophy:** "There should be one—and preferably only one—obvious way to do it."

Major Versions:

- **Python 2:** Introduced in 2000, legacy now.
- **Python 3:** Launched in 2008, is the standard (not backward-compatible with Python 2).
- **Current versions:** Always recommend using the latest Python 3.x.

Fun fact: Python is now maintained by the Python Software Foundation.

---

3. Python's Design Principles

- Emphasizes **clarity and readability** ("beautiful is better than ugly").
- Code is meant to be easy for humans to read, not just computers.
- Encourages using whitespace and indentation for structure, unlike many other languages.

Zen of Python (Easter Egg!): Type `import this` in Python interpreter to see guiding aphorisms.

Example Output:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
```

```
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

---

4. PEP: Python Enhancement Proposals

- **PEP** stands for **Python Enhancement Proposal**.
- Documents that propose features, design, style guidelines, and improvements.
- **PEP 8:** The official style guide for writing Python code.
- **PEP 20:** The Zen of Python (see above).

Why follow PEP 8?

- Keeps code consistent, readable, and maintainable.
- Encourages good habits from the start.

Key PEP 8 Rules (with examples):

Bad Practice	Good Practice
`x=10`	`x = 10`
`def myFunction():`	`def my_function():`
`if(x>0): print(x)`	`if x > 0:\n  print(x)`
80+ character lines	Wrap lines before 80 chars

Example:

```python
# PEP 8 compliant
def calculate_sum(a, b):
    return a + b
```

---

5. Indentation: Python's Structure

- **Indentation replaces curly braces or keywords (like "end")** used in other languages.
- Blocks of code are defined by **consistent indentation** (default: 4 spaces).
- **Mixing spaces and tabs is discouraged** (use spaces).

Why is this important?

- Python will throw an `IndentationError` if code is not properly indented.

Example:

```python
# Correct
if 10 > 2:
    print("10 is greater than 2")
```

```
    print("This is inside the if block.")

print("This is outside the if block.")

# Incorrect (will raise error)
if 10 > 2:
print("10 is greater than 2")  # IndentationError!
```

Best Practice:

- Use **4 spaces** for each level of indentation (can be set in your IDE settings).

---

6. Quick Comparison: Indentation vs. Other Languages

JavaScript Example:

```
if (x > 0) {
    console.log(x);
}
```

Python Example:

```
if x > 0:
    print(x)
```

Notice: No `{}` in Python—just indentation.

---

7. Quick Activity for Students

- **Try in your IDE:**
    1. Remove the indentation in a code block and observe the error.
    2. Re-indent and see it work.

---

8. Summary Table

Concept	Python Example	Notes
Variable Name	`snake_case` (e.g., `my_variable`)	Lowercase with underscores
Indentation	`if x > 0:\n    print(x)`	4 spaces per block
Line Length	`# Keep lines < 80 chars`	Improves readability
Comments	`# This is a comment`	Use `#` for inline comments

Python's Global Community

- Python is open source and globally supported.
- **Official docs:** docs.python.org
- Many conferences: PyCon, EuroPython, etc.

Iv. Python Syntax, Variables, And Data Types

IV. Python Syntax, Variables, and Data Types

---

1. Python Syntax: The Basics

Case Sensitivity

- Python distinguishes between uppercase and lowercase letters.
  - `age` and `Age` are two different variables.

Line Endings

- Each statement typically ends with a newline (no semicolons needed, but allowed).
- Multiple statements can be placed on one line with `;` (not recommended).

Comments

- Single-line comments use `#`.
- Multi-line comments use triple quotes `""" ... """` (also used for docstrings).

Example:

```
# This is a comment
name = "Sam"  # This is also a comment

"""
This is a
multi-line comment or docstring.
"""
```

---

2. Variables in Python

Dynamic Typing

- You don't need to declare a variable's type.
- The type is determined by the value assigned, and you can change it at runtime.
- **In theory:** This is called *dynamic typing*. It contrasts with *static typing* in languages like Java or C++.

Variable Declaration and Assignment

```
x = 5          # int
x = "hello"    # Now x is a str!
```

Variable Naming Rules

- Must start with a letter or underscore.
- Can include letters, numbers, and underscores.
- Cannot be a Python keyword (like `if`, `for`, `class`, etc.).

Example:

```
_age = 25
user2 = "Alice"
# 2user = "error"  # Invalid!
```

Why Dynamic Typing?

- Makes code more flexible and quick to write.
- Encourages prototyping and experimentation.
- **Downside:** Can lead to bugs if you're not careful (e.g., accidentally treating a string as a number).

---

3. Data Types in Python

a) Numeric Types

- **int** – Integer numbers

  ```
  a = 10
  b = -23
  ```

- **float** – Floating point (decimal) numbers

  ```
  pi = 3.14159
  price = -2.75
  ```

- **complex** – Complex numbers (less common)

  ```
  z = 2 + 3j
  ```

b) Text Type

- **str** – String (sequence of Unicode characters)

  ```
  message = "Hello, World!"
  char = 'A'
  ```

c) Boolean Type

- **bool** – Logical values `True` or `False`

  ```
  is_valid = True
  is_finished = False
  ```

d) Sequence Types

- **list** – Ordered, mutable sequence

```
fruits = ["apple", "banana", "cherry"]
fruits[0] = "grape"  # Lists are mutable!
```

- **tuple** – Ordered, immutable sequence

```
point = (2, 3)
# point[0] = 5  # Error! Tuples are immutable.
```

e) Mapping Type

- **dict** – Key-value pairs (dictionary)

```
student = {"name": "Sam", "age": 21}
print(student["name"])  # "Sam"
```

f) Set Types

- **set** – Unordered, unique elements

```
colors = {"red", "green", "blue"}
colors.add("yellow")
```

---

4. Type Checking and Conversion

- Use `type()` to check a variable's type.

```
x = 5
print(type(x))  # <class 'int'>
```

- Convert between types:

```
x = "123"
y = int(x)      # y is now 123 (int)
z = float(x)    # z is now 123.0 (float)
name = str(123)  # name is "123" (str)
```

---

5. Differences from Other Languages

Feature	Python	Java/C++
Type Declaration	`x = 10`	`int x = 10;`
Type Changes Allowed?	Yes	No (fixed at declaration)
End of Statement	Newline	Semicolon (`;`)
Block Delimiters	Indentation	`{ }` braces
Variable Name	Dynamic (no type prefix)	Static (type required)

---

6. Examples for Students

Variable Assignments and Types

```
age = 18           # int
height = 1.75      # float
name = "Charlie"   # str
is_student = True  # bool
```

Lists and Dictionaries

```
grades = [90, 85, 88, 92]  # list of ints
student_info = {"name": "Sam", "id": 12345}
```

Type Conversion

```
# Input always returns str!
user_age = input("Enter your age: ")
age_num = int(user_age)
```

---

7. Quick Activity / Demo

- Try assigning different types to the same variable.
- Try using `type()` to print out the types.
- Try making a list, tuple, dict, and set in your IDE.

---

8. Summary Table

Type	Example	Mutable?	Ordered?	Unique?
int	`a = 5`	N/A	N/A	N/A
float	`b = 3.14`	N/A	N/A	N/A
str	`s = "hi"`	No	Yes	N/A
bool	`is_ready = False`	N/A	N/A	N/A
list	`l = [1, 2, 3]`	Yes	Yes	No
tuple	`t = (1, 2, 3)`	No	Yes	No
dict	`d = {"a":1, "b":2}`	Yes	No	Keys: Yes
set	`s = {1, 2, 3}`	Yes	No	Yes

---

9. Pro Tips

- Use meaningful variable names.
- Start with lowercase, use underscores for multiple words (`user_name`).
- Avoid using reserved keywords (like `class`, `if`, `def`) as variable names.

LESSON 1: INTRODUCTION TO PYTHON BASICS

Learning Objectives

By the end of this lesson, students will be able to:

1. Set up a Python development environment
2. Understand Python history and its syntax conventions
3. Write simple Python programs using variables, basic data types, and I/O functions
4. Begin to follow PEP8 formatting guidelines

1. Setting Up the Python Environment (15 minutes)

Topics:

```
1. Installing Python (https://www.python.org/downloads/)
1. Choosing an IDE:
    1 . VS Code (recommended for beginners and professionals)
    2. PyCharm (feature-rich, great for larger projects)
    3. Jupyter Notebooks (interactive, great for data science)
    4. Thoony
    5. Spyder
```

1. Running Python code:
    - Using terminal or command prompt
    - Using Python shell
    - Running .py files
    - Inline execution in Jupyter

2. Introduction to Python

Brief history of Python

```
Created by Guido van Rossum in the late 1980s

Released in 1991

Emphasis on readability and simplicity
```

Why Python?

```
Popularity

Ease of learning

Wide applications (web dev, data science, scripting, etc.)
```

PEP8 and Code Style:

```
Naming conventions
```

`Indentation (4 spaces)`

`Readable code`

Readme

Python Programming

01 - Intro

V. Input:Output Functions, Print Statements, And Console Communication

V. Input/Output Functions, Print Statements, and Console Communication (20 minutes)

---

1. Printing Output with `print()`

Purpose:

- The `print()` function displays information on the console (standard output).

Basic Usage:

```
print("Hello, World!")
```

Output:

```
Hello, World!
```

Printing Multiple Items:

- You can print several values separated by commas. Python will add a space by default.

```
name = "Alice"
age = 23
print("Name:", name, "Age:", age)
```

Output:

```
Name: Alice Age: 23
```

String Concatenation:

- You can use `+` to join strings.

```
print("Hello, " + name + "!")
```

Output:

```
Hello, Alice!
```

> Note: All arguments must be strings when using `+`. If not, you need to convert them with `str()`.

Advanced Printing: Formatting Strings

- **f-strings** (Python 3.6+): Easy way to insert variable values.

```
print(f"My name is {name} and I am {age} years old.")
```

Output:

```
My name is Alice and I am 23 years old.
```

- **`format()` method:**

```
print("My name is {} and I am {} years old.".format(name, age))
```

Custom Endings and Separators:

- **`end` argument:** Change what is printed at the end (default is newline `\n`).

```
print("Hello,", end=" ")
print("World!")
```

Output:

```
Hello, World!
```

- **`sep` argument:** Change how multiple arguments are separated (default is space).

```
print("2025", "05", "19", sep="-")
```

Output:

```
2025-05-19
```

---

2. Receiving Input with `input()`

Purpose:

- `input()` lets you get user input from the console.
- It always returns a string.

Basic Usage:

```
name = input("What is your name? ")
print("Hello,", name)
```

Sample Run:

```
What is your name? Sam
Hello, Sam
```

Converting Input to Other Types

- Since `input()` always returns a string, you need to **convert** it for numbers:

```
age = input("How old are you? ")
print(type(age))  # <class 'str'>

# Convert to int
age = int(age)
print(type(age))  # <class 'int'>
```

Example: Adding Two Numbers from User Input

```
num1 = input("Enter a number: ")
num2 = input("Enter another number: ")

# Convert to numbers
num1 = float(num1)
num2 = float(num2)

print("The sum is:", num1 + num2)
```

Sample Run:

```
Enter a number: 3
Enter another number: 4.5
The sum is: 7.5
```

Error to Watch For: If you try to convert something that's not a number:

```
num = int("abc")  # This will cause a ValueError!
```

---

3. Console Communication – Tips and Best Practices

- **Prompt clearly:** Always give the user instructions on what to enter.
  - Good: `input("Enter your age: ")`
  - Bad: `input()`
- **Use print() to make output user-friendly and informative.**
- **Always convert user input** when a number is needed (`int()` or `float()`).

---

4. Demo Activities for Students

Activity 1: Greet the user

```
name = input("What's your name? ")
print("Welcome,", name, "!")
```

Activity 2: Simple calculator

```
a = float(input("First number: "))
b = float(input("Second number: "))
```

```
print(f"Sum: {a + b}")
print(f"Product: {a * b}")
```

Activity 3: Personalized output

```
color = input("What is your favorite color? ")
animal = input("What is your favorite animal? ")
print(f"Your favorite color is {color} and your favorite animal is {animal}.")
```

---

5. Summary Table

Function	Purpose	Example Usage	Returns
`print()`	Output to console	`print("Hello")`	None
`input()`	Get input from user	`name = input("Name: ")`	`str`

---

6. Pro Tip

- If you want to print a blank line, just use:

  ```
  print()
  ```

- For complex inputs, you can split and process:

  ```
  data = input("Enter two numbers separated by space: ")
  x, y = data.split()
  x = int(x)
  y = int(y)
  print(x, y)
  ```

Let me know if you want exercises, a handout, or an interactive Jupyter notebook for this section!

Vi. Hands-On Coding- Simple Python Programs

VI. Hands-on Coding: Simple Python Programs (20 minutes - work)

Program 1: Greet the User

Objective:

- Practice input/output, variables, and string formatting.

Step-by-Step:

1. **Prompt the user** for their name using `input()`.
2. **Store** the name in a variable.
3. **Print** a personalized greeting using `print()`.

Example Code:

```
# Ask for the user's name
name = input("What is your name? ")

# Greet the user
print("Hello,", name + "! Welcome to Python programming.")
```

Explanation:

- The `input()` function displays the prompt and waits for the user to type their name.
- The entered name is stored in the variable `name`.
- The greeting uses string concatenation and the `print()` function to display a message.

Program 2: Add Two Numbers Entered by the User

Objective:

- Practice input, type conversion, variables, arithmetic operations, and output formatting.

Step-by-Step:

1. **Prompt the user** to enter two numbers.
2. **Store** both inputs in variables.
3. **Convert** the inputs from strings to floats or ints.
4. **Add** the two numbers.
5. **Print** the result.

Example Code:

```
# Ask the user for two numbers
num1 = input("Enter the first number: ")
num2 = input("Enter the second number: ")

# Convert the inputs to float (to allow decimals)
num1 = float(num1)
num2 = float(num2)

# Add the numbers
result = num1 + num2

# Print the result using an f-string
print(f"The sum of {num1} and {num2} is {result}.")
```

Explanation:

- `input()` collects user input as strings.
- `float()` converts input to numbers (for whole numbers, `int()` can also be used).
- The sum is stored in `result`.
- The output is displayed using an **f-string** for clarity and readability.

Bonus Program: Mini Calculator (Combines Both Examples and More)

Objective:

- Reinforce input, output, data types, variables, and arithmetic.
- Demonstrate user-friendly interaction.

Example Code:

```
# Welcome message
print("Welcome to the Python Mini Calculator!")

# Get user's name
name = input("What's your name? ")
print(f"Hi, {name}! Let's do some math.")

# Get two numbers from user
a = float(input("Enter the first number: "))
b = float(input("Enter the second number: "))

# Perform calculations
print(f"{a} + {b} = {a + b}")
print(f"{a} - {b} = {a - b}")
print(f"{a} * {b} = {a * b}")
if b != 0:
    print(f"{a} / {b} = {a / b}")
else:
    print("Cannot divide by zero.")
```

Explanation:

- This example brings together everything from today: input/output, variables, data types, arithmetic, conditionals (for division by zero), and string formatting.

Activity Instructions for Students

- Type each example into your IDE or Jupyter notebook.
- Modify the programs (change variable names, messages, or add new features).
- Experiment with both `int` and `float` conversions.
- Try to break the program—what happens if you type a word instead of a number?

Summary of Skills Practiced

- Declaring and using variables
- Getting user input (`input()`)
- Printing output (`print()`)
- Using data types (`str`, `int`, `float`)
- Performing arithmetic operations

- String formatting (`+` and f-strings)
- Handling basic errors (like dividing by zero)

Vii. Q&A And Wrap-Up

VII. Q&A and Wrap-Up (10 minutes)

---

1. Recap of Key Points

Let's quickly review what we covered today:

a) Python Environment Setup

- You learned how to download and install Python, and use different IDEs: PyCharm, VS Code, Thonny, and Jupyter Notebooks.
- You ran your first Python program: `print("Hello, World!")`

b) Python Language Basics

- **Brief History:** Python was created by Guido van Rossum and emphasizes readability and simplicity.
- **PEP Standards:** Especially PEP 8 for coding style, which encourages writing clean, consistent code.
- **Indentation:** Python uses indentation to define code blocks, which is different from many other programming languages.

c) Variables and Data Types

- You created variables without declaring their type first (dynamic typing).
- Covered core data types: `int`, `float`, `str`, `bool`, `list`, `tuple`, `dict`, `set`.

d) Input and Output

- Used `input()` to get user data (always returns a string).
- Used `print()` for output, including formatting output with f-strings.

e) Hands-on Coding

- Wrote simple programs to greet the user and add two numbers.
- Practiced combining variables, input/output, and arithmetic.

---

2. Encourage Questions and Discussion

Open the Floor:

- "Do you have any questions about anything we covered today?"
- "Is there anything you found confusing or would like another example of?"
- "Are you curious about where Python can take you beyond these basics?"

Tips for Teachers:

- Give students a minute to think, or suggest a think-pair-share with a neighbor.
- Encourage "there's no such thing as a silly question"—everyone starts somewhere!
- If time allows, consider a "rapid-fire Q&A": Ask students to shout out things they remember from today, and briefly elaborate on each.

---

3. Share Learning Resources

Point students toward reputable sources for continued learning and self-study:

a) Official Documentation

- Python.org – Download, documentation, news, community links.

b) Beginner-Friendly Tutorials

- W3Schools Python Tutorial – Step-by-step basics, examples, and practice exercises.
- Real Python – Great articles, tutorials, and guides.

c) Coding Standards

- PEP 8 - Style Guide for Python Code – Essential for writing professional and readable Python code.

d) Interactive Learning

- Replit Online Python IDE – Practice coding online.
- Google Colab – Free Jupyter notebooks in the cloud.

e) Community and Help

- Stack Overflow – Ask questions, find answers.
- Python Discord – Chat, get help, join code events.

---

4. Suggestions for Next Steps

- **Practice:** Try modifying today's programs—change messages, do other calculations, or ask for different user information.
- **Preview for Next Lesson:** We'll explore more complex data structures and control flow (like if statements and loops).
- **Homework Reminder:** (Refer to the assignment you gave earlier.)

---

5. End with Motivation

> "Everyone starts with the basics! Don't worry if it feels new or a bit overwhelming—practice is the key. With each lesson, you'll become more comfortable and capable as a Python programmer."

---

6. Thank You and See You Next Time!

- "Thank you for your attention and participation today!"
- "See you next class—keep coding!"

Lesson 02

Ii. Operators, Counting, Flow Control, Enumerators

II. Operators, Counting, Flow Control, Enumerators

A. Operators in Python

Operators are symbols that perform operations on variables and values. In Python, the main types are:

1. Arithmetic Operators

Operator	Name	Example	Result
+	Addition	`3 + 2`	5
-	Subtraction	`7 - 4`	3
*	Multiplication	`5 * 2`	10
/	Division	`10 / 4`	2.5 (float)
//	Floor Division	`10 // 4`	2 (integer)
%	Modulo	`10 % 3`	1
**	Exponentiation	`2 ** 3`	8

Examples:

```
print(5 + 3)        # 8
print(7 / 2)        # 3.5
print(7 // 2)       # 3
print(10 % 4)       # 2
print(2 ** 4)       # 16
```

Common Mistake: Using `/` when you want integer division:

```
print(9 / 2)   # 4.5 (float) CORRECT
print(9 // 2)  # 4 (integer division) CORRECT
```

```
# INCORRECT:
result = 9 // 2.0   # Result is float 4.0, not integer, as one operand is float
```

2. Comparison Operators Used to compare values. Always returns `True` or `False`.

Operator	Description	Example	Result
==	Equal	`3 == 4`	False
!=	Not Equal	`5 != 3`	True
<, >	Less, Greater	`2 < 5`	True
<=, >=	Less/Greater or Equal	`5 >= 5`	True

Examples:

```
a = 5
b = 7
print(a < b)        # True
print(a == b)       # False
print(a != b)       # True
```

Common Mistake: Assigning (`=`) instead of comparing (`==`):

```
# INCORRECT:
if a = b:
    print("Equal")  # SyntaxError

# CORRECT:
if a == b:
    print("Equal")
```

3. Logical Operators Combine multiple conditions.

- `and` – True if both are True
- `or` – True if at least one is True
- `not` – Inverts the condition

Examples:

```
x = 10
print(x > 5 and x < 20)    # True
print(x < 5 or x > 5)      # True
print(not (x == 10))       # False
```

B. Counting with `range()`

What is `range()`?

- `range(start, stop, step)` generates a sequence of numbers.
- Often used with `for` loops.

Common Usages:

```
# Count from 0 to 4 (default start is 0)
for i in range(5):
    print(i)   # 0, 1, 2, 3, 4

# Count from 1 to 5
for i in range(1, 6):
    print(i)   # 1, 2, 3, 4, 5

# Count from 2 to 10, stepping by 2
for i in range(2, 11, 2):
    print(i)   # 2, 4, 6, 8, 10
```

Counting Downwards:

```
for i in range(10, 0, -2):
    print(i)  # 10, 8, 6, 4, 2
```

Common Mistakes:

- Forgetting that the `stop` value is **exclusive** (not included).

  ```
  for i in range(1, 5):
      print(i)   # 1, 2, 3, 4 (not 5)
  ```

- Using a negative step without adjusting start/stop.

  ```
  for i in range(5, 0, -1):
      print(i)  # 5, 4, 3, 2, 1
  ```

Other Useful Counting:

- **Sum all numbers from 1 to N:**

  ```
  total = 0
  for i in range(1, 11):
      total += i
  print(total)  # 55
  ```

- **Generate a list of numbers:**

  ```
  numbers = list(range(5))
  print(numbers)  # [0, 1, 2, 3, 4]
  ```

C. Enumerators (`enumerate()`)

What is `enumerate()`?

- Lets you loop over a sequence **and** get the index and value at the same time.

Example:

```
colors = ['red', 'green', 'blue']
for index, color in enumerate(colors):
    print(f"Index: {index}, Color: {color}")
```

Output:

```
Index: 0, Color: red
Index: 1, Color: green
Index: 2, Color: blue
```

Correct Usage:

- Using both index and value for display, calculations, or logic:

  ```
  for idx, val in enumerate(['a', 'b', 'c']):
      print(idx, val)
  ```

Customizing Start Index:

```
for idx, val in enumerate(['apple', 'banana'], start=1):
    print(idx, val)
# Output: 1 apple 2 banana
```

Incorrect Usage:

- Using `enumerate()` on something not iterable (e.g., a number):

  ```
  # INCORRECT:
  for idx, val in enumerate(100):
      print(idx, val)  # TypeError: 'int' object is not iterable
  ```

- Forgetting to unpack both index and value:

  ```
  # INCORRECT:
  for item in enumerate(['a', 'b']):
      print(item)  # Prints tuples: (0, 'a'), (1, 'b')
  # CORRECT:
  for idx, val in enumerate(['a', 'b']):
      print(idx, val)
  ```

Similar/Related Cases:

- **Looping with range(len(list)):** (Not recommended unless you need the index specifically)

  ```
  mylist = ['x', 'y', 'z']
  for i in range(len(mylist)):
      print(i, mylist[i])
  ```

- **Using zip() for two lists:**

  ```
  names = ['Alice', 'Bob']
  scores = [85, 90]
  ```

```
for name, score in zip(names, scores):
    print(name, score)
```

Summary Table

Feature	Usage Example	Notes/Corrections
Arithmetic	`a + b`, `a // b`	Use `//` for int division
Comparison	`a == b`, `a < b`	Use `==`, not `=` for compare
Logical	`x > 2 and x < 10`	Combine multiple conditions
Range	`for i in range(3, 8, 2): ...`	Step can be negative
Enumerate	`for idx, item in enumerate(list): ...`	Unpack both index and value

Let us check play with:

1. **Arithmetic & Comparison:**
   - Write a program that asks for two numbers and prints their sum, difference, and whether they are equal.
2. **Counting:**
   - Print all even numbers between 20 and 30.
   - Sum all numbers from 50 down to 10 (inclusive), stepping by -5.
3. **Enumerate:**
   - Print a shopping list with item numbers starting at 1.

Iii. Conditional Statements

III. Conditional Statements

Conditional statements let your program make decisions and perform actions depending on different conditions.

A. The `if` Statement

Basic Structure:

```
if condition:
    # code to execute if condition is True
```

Example:

```
x = 10
if x > 5:
    print("x is greater than 5")
```

Output:

```
x is greater than 5
```

How it Works:

- The code inside the `if` block (indented) runs only if the `condition` evaluates to `True`.
- Indentation (typically 4 spaces) is crucial in Python to define code blocks.

Correct Usage:

```
temperature = 30
if temperature > 25:
    print("It's a hot day.")
```

Incorrect Usage:

```
# INCORRECT: No colon
if temperature > 25
    print("It's a hot day.")  # SyntaxError

# INCORRECT: No indentation
if temperature > 25:
print("It's a hot day.")      # IndentationError

# INCORRECT: Assignment instead of comparison
x = 10
if x = 5:
    print("x is 5")           # SyntaxError, should be == for comparison
```

Other Valid Conditions:

- Any expression that evaluates to `True` or `False`, including:
  - Comparisons (`>`, `<`, `==`)
  - Checking membership: `if "a" in "apple":`
  - Boolean variables

B. The `else` Statement

`else` defines what to do if the `if` condition is not met.

Structure:

```
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 are too young to vote.")
```

Output:

```
You are too young to vote.
```

Correct Usage:

- `else` must always follow an `if` (and any `elif`).
- No condition after `else`.

Incorrect Usage:

```
# INCORRECT: Adding a condition to else
if age >= 18:
    print("Adult")
else age < 18:
    print("Minor")    # SyntaxError

# CORRECT:
if age >= 18:
    print("Adult")
else:
    print("Minor")
```

C. The `elif` Statement

`elif` stands for "else if", allowing multiple exclusive conditions.

Structure:

```
if condition1:
    # code if condition1 is True
elif condition2:
    # code if condition2 is True
elif condition3:
    # code if condition3 is True
else:
    # code if none above are True
```

Example:

```python
score = 72
if score >= 90:
    print("Grade A")
elif score >= 80:
    print("Grade B")
elif score >= 70:
    print("Grade C")
else:
    print("Grade D")
```

Output:

```
Grade C
```

Multiple Conditions:

- Python checks conditions **top-down**. The first `True` one runs, the rest are skipped.

Correct Usage:

- Only one branch is ever executed in an `if-elif-else` block.

Incorrect Usage:

```python
# INCORRECT: More than one block can run (NOT in if/elif/else)
if score > 70:
    print("Pass")
if score > 90:
    print("Excellent")  # Both can run if score > 90

# CORRECT: Use elif/else for mutually exclusive
if score > 90:
    print("Excellent")
elif score > 70:
    print("Pass")
else:
    print("Try again")
```

Omitting `else`:

- `else` is optional; if omitted and no condition is met, nothing happens.

Edge Cases & Truthy/Falsy Values

- Any value in Python can be tested in a condition.
  - Zero, empty string/list/tuple/dict, and `None` are considered **False**.
  - Non-empty, non-zero values are **True**.

Examples:

```python
if []:
    print("Non-empty")   # Will NOT print

if [1, 2, 3]:
    print("Non-empty list")  # Will print

if 0:
    print("Zero")   # Will NOT print

if "Hello":
    print("Has text")  # Will print
```

Similar Cases / Alternatives to Conditional Statements

1. Nested `if` Statements

```python
num = 5
if num > 0:
    if num < 10:
        print("Number is between 1 and 9")
```

2. Multiple Separate `if` Statements (Not mutually exclusive; all can run)

```python
num = 7
if num > 5:
    print("Greater than 5")
if num % 2 == 1:
    print("Odd number")
```

3. Ternary Conditional Expression ("one-line if")

```python
status = "Adult" if age >= 18 else "Minor"
print(status)
```

4. No `switch/case` in Basic Python

- Python doesn't have a built-in `switch/case` like some languages (Java, C). Use `if/elif/else` or dictionaries for similar logic.

Example of switch-case alternative:

```python
choice = 2
if choice == 1:
    print("Option 1 selected")
elif choice == 2:
    print("Option 2 selected")
elif choice == 3:
    print("Option 3 selected")
```

```
else:
    print("Invalid option")

# Or, with dictionary mapping:
def option_one():
    print("Option 1 selected")
def option_two():
    print("Option 2 selected")
def option_three():
    print("Option 3 selected")

options = {1: option_one, 2: option_two, 3: option_three}
options.get(choice, lambda: print("Invalid option"))()
```

More alternatives: https://docs.python.org/3/tutorial/controlflow.html

In Python 3.10 and above, the `match` statement is used for pattern matching (similar to switch-case in other languages). It allows you to compare a value against several patterns and execute code based on which pattern matches.

```
fruit = "apple"

match fruit:
    case "apple":
        print("It's an apple!")
    case "banana":
        print("It's a banana!")
    case "orange":
        print("It's an orange!")
    case _:
        print("Unknown fruit")
```

D. Example Program: Check if a Number is Positive, Negative, or Zero

```
num = float(input("Enter a number: "))

if num > 0:
    print("The number is positive.")
elif num < 0:
    print("The number is negative.")
else:
    print("The number is zero.")
```

Common Pitfalls and Corrections:

- **Not using `elif`, causing redundant checks:**

```
if num > 0:
    print("Positive")
if num == 0:
    print("Zero")
if num < 0:
    print("Negative")
# More than one could run if logic is off!
```

- **Using `else` without `if`:**

```
# INCORRECT
else:
    print("Error")  # SyntaxError
```

- **Missing indentation:**

```
if num > 0:
print("Positive")  # IndentationError
```

Practice and check our knowledge:

1. Write a program to print if a person is a "teenager", "adult", or "child" based on their age.
2. Check if a given year is a leap year.
3. Use a ternary expression to print "Even" or "Odd" for a given number.

Summary Table

Statement	Usage Example	Notes/Corrections
`if`	`if a > b:`	Condition must return `True`/`False`
`else`	`else:`	No condition after else
`elif`	`elif score >= 70:`	Use for multiple, exclusive options
Ternary	`x = "yes" if flag else "no"`	One-line assignment based on condition
No switch	Use if/elif/else or dict for switch	No native switch/case in Python

---

Iv. Loops- For And While, Break And Continue

IV. Loops: for and while, break and continue

A. For Loops

What is a `for` loop? A `for` loop repeats code for each item in a sequence (such as a list, tuple, string, or range).

Basic Structure:

```
for variable in sequence:
    # code block
```

Examples: Iterate over a list:

```
fruits = ['apple', 'banana', 'cherry']
for fruit in fruits:
    print(fruit)
# Output:
# apple
# banana
# cherry
```

Iterate over a string:

```
for char in "Python":
    print(char)
# Output:
# P y t h o n (one per line)
```

Use `range()` for counting:

```
for i in range(1, 6):
    print(i)
# Output: 1 2 3 4 5
```

Iterate with both index and value using `enumerate()`:

```
colors = ['red', 'green', 'blue']
for idx, color in enumerate(colors):
    print(idx, color)
```

Correct Usage:

- Looping over any iterable (list, tuple, set, dictionary, string, etc.)

Incorrect Usage:

- Trying to loop over something that isn't iterable:

  ```
  # INCORRECT:
  for x in 5:
      print(x)  # TypeError: 'int' object is not iterable
  ```

- Forgetting the colon or indentation:

  ```
  # INCORRECT:
  for fruit in fruits
      print(fruit)  # SyntaxError
  ```

- Modifying a list while iterating over it (can lead to skipped items or errors):

```
mylist = [1, 2, 3]
for item in mylist:
    if item == 2:
        mylist.remove(item)
# List modification during iteration is unsafe.
```

Other Sequence Types:

- **Dictionaries:** Looping through keys, values, or both:

```
mydict = {'a': 1, 'b': 2}
for key in mydict:
    print(key, mydict[key])
for value in mydict.values():
    print(value)
for key, value in mydict.items():
    print(key, value)
```

B. While Loops

What is a `while` loop? Repeats code **as long as** a condition is `True`. Great for situations when you don't know ahead of time how many times to repeat.

Basic Structure:

```
while condition:
    # code block
```

Examples: Count up to 5:

```
count = 1
while count <= 5:
    print(count)
    count += 1
# Output: 1 2 3 4 5
```

User input until they guess the correct answer:

```
secret = 7
guess = None
while guess != secret:
    guess = int(input("Guess the number: "))
print("You got it!")
```

Correct Usage:

- When the number of iterations isn't known beforehand.

Incorrect Usage:

- **Forgetting to update the condition variable (infinite loop):**

```
# INCORRECT:
i = 0
while i < 3:
    print(i)
# This will loop forever because i is never incremented!
```

- **Infinite loop without a break:**

```
while True:
    print("Runs forever")  # Must have a break condition inside!
```

Common Mistakes with `while`:

- Using assignment `=` instead of comparison `==`:

```
# INCORRECT:
while i = 5:
    print(i)  # SyntaxError
```

- Not initializing loop variables.

C. `break` and `continue`

break: Immediately stops the nearest enclosing loop.

Example:

```
for i in range(10):
    if i == 5:
        break
    print(i)
# Output: 0 1 2 3 4
```

While Loop Example:

```
while True:
    num = int(input("Enter a positive number (or -1 to quit): "))
    if num == -1:
        break
    print("You entered:", num)
```

---

continue: Skips the rest of the code **inside the loop for that iteration** and moves to the next loop cycle.

Example:

```
for i in range(1, 6):
    if i % 2 == 0:
        continue  # Skip even numbers
    print(i)
# Output: 1 3 5
```

While Loop Example:

```
count = 0
while count < 5:
    count += 1
    if count == 3:
        continue
    print(count)
# Output: 1 2 4 5 (skips 3)
```

Correct Usage:

- Use `break` to exit a loop early (e.g., searching for an item).
- Use `continue` to skip certain values (e.g., skipping over unwanted data).

Incorrect Usage:

- Using `break` or `continue` outside a loop:

```
# INCORRECT:
if x == 5:
    break   # SyntaxError
```

Similar and Related Cases in Python Loops:

1. else with Loops

- Python allows an `else` block **after loops**. The `else` part runs **only if the loop didn't exit with a `break`**.

Example:

```
for n in range(2, 10):
    if n % 7 == 0:
        print("Found number divisible by 7:", n)
        break
else:
    print("No number divisible by 7 found.")
# If no break, the else runs.
```

2. Nested Loops

- Loops inside other loops, e.g., to print tables or work with multidimensional data.

```
for row in range(1, 4):
    for col in range(1, 4):
        print(row, col)
```

3. Looping Over Multiple Sequences (zip):

```
names = ["Alice", "Bob"]
scores = [85, 90]
for name, score in zip(names, scores):
    print(name, score)
```

4. List Comprehensions (advanced preview):

- A more "Pythonic" way to loop and build lists.

```
squares = [x**2 for x in range(1, 6)]
print(squares)  # [1, 4, 9, 16, 25]
```

Common Pitfalls and Corrections:

- **Infinite loops:**
  - Not updating the loop variable or using a `while True:` without a break.
- **Off-by-one errors:**
  - Be careful with `range()`—it's exclusive at the stop value.
- **Changing a list while iterating over it:**
  - Can result in skipping items or unexpected behavior.
  - Instead, loop over a copy: `for item in mylist[:]: ...`

Let's do some coding:

1. **For Loop:** Print all even numbers between 1 and 20.
2. **While Loop:** Ask the user to enter numbers until they type 0. Output the sum.
3. **break:** Loop through a list of names, stop if you find "Alice".
4. **continue:** Loop through numbers 1 to 10, print all except multiples of 3.
5. **Nested Loops:** Print a multiplication table (1–5).

Summary Table

Statement	Usage Example	Notes/Corrections
for	`for x in [1,2,3]: print(x)`	Loop over any iterable
while	`while x < 10: x += 1`	Update loop variable!
break	`if found: break`	Exits loop immediately

Statement	Usage Example	Notes/Corrections
continue	`if not ok: continue`	Skips rest of current iteration
else (loops)	`for ... else:`	Runs if no break occurred

V. Combined Hands-On Assignment- Python Control Flow Mini-Project

V. Combined Hands-On Assignment: Python Control Flow Mini-Project

Assignment Description

Create a Python program that **assists a small club in managing some of their daily tasks.** Your program should be menu-driven (using loops and conditional statements) and let the user choose from several options. Each option demonstrates a concept from this lesson.

Menu Options and Requirements

The program should display a menu like this:

```
Please select an option:
1. Show all numbers from 1 to 20 divisible by 3
2. Calculate the sum of all numbers from 1 to 100
3. Enter numbers until 0 is typed, then display their sum
4. Filter a list of names to show those starting with a vowel
5. Print a triangle pattern of stars with user-defined height
0. Exit
```

- The program must **keep running** (loop) until the user selects "0. Exit".
- After each operation, the menu should reappear.

Detailed Breakdown of Each Option

Option 1: Show numbers from 1 to 20 divisible by 3

- Use a `for` loop and conditional (`if`) to check divisibility.

Option 2: Sum numbers from 1 to 100

- Use a `for` loop and an accumulator variable.

Option 3: Enter numbers until 0 is typed; display the sum

- Use a `while` loop and conditional logic.
- Input validation is encouraged.

Option 4: Print names starting with a vowel

- Prompt the user to enter a comma-separated list of names.
- Use a `for` loop, `if`, and string methods.

Option 5: Print triangle pattern

- Prompt the user for the triangle's height.
- Use nested `for` loops to print a left-aligned triangle of stars (`*`).

Sample Program Structure

```
def print_menu():
    print("\nPlease select an option:")
    print("1. Show all numbers from 1 to 20 divisible by 3")
    print("2. Calculate the sum of all numbers from 1 to 100")
    print("3. Enter numbers until 0 is typed, then display their sum")
    print("4. Filter a list of names to show those starting with a vowel")
    print("5. Print a triangle pattern of stars with user-defined height")
    print("0. Exit")

while True:
    print_menu()
    choice = input("Enter your choice (0-5): ")

    if choice == "1":
        print("Numbers from 1 to 20 divisible by 3:")
        for i in range(1, 21):
            if i % 3 == 0:
                print(i, end=' ')
        print()

    elif choice == "2":
        total = 0
        for i in range(1, 101):
            total += i
        print("The sum of numbers from 1 to 100 is:", total)

    elif choice == "3":
        total = 0
        while True:
            num = int(input("Enter a number (0 to finish): "))
            if num == 0:
                break
            total += num
        print("The sum of entered numbers is:", total)

    elif choice == "4":
        names_input = input("Enter names separated by commas: ")
        names = [name.strip() for name in names_input.split(",")]
        print("Names starting with a vowel:")
        for name in names:
```

```
            if name and name[0].lower() in "aeiou":
                print(name)

    elif choice == "5":
        height = int(input("Enter the height of the triangle: "))
        print("Triangle pattern:")
        for i in range(1, height+1):
            print("*" * i)

    elif choice == "0":
        print("Goodbye!")
        break

    else:
        print("Invalid choice. Please select from 0 to 5.")
```

Instructor Notes & Learning Objectives

- This assignment reinforces **loops** (for, while), **conditionals** (if, elif, else), **input/output**, **operators**, **enumerate/string operations**, and **nested logic**.
- Encourages thinking about **program structure**, user interaction, and combining multiple skills.
- You can encourage students to use **comments** and experiment with error-handling (e.g., handling non-integer input).

Extension Ideas (for advanced students or extra credit):

- Add input validation for all inputs.
- Allow repeating option 4 with different name lists until a blank line is entered.
- Enhance the triangle pattern to be centered or inverted.
- Add a summary at the end of each loop iteration showing which options have been run.

Vi.Wrap Up And Homework

VI. Wrap Up & Homework Assignment

A. Lesson Wrap Up

Key Takeaways:

- **Operators** are used for math and comparisons.
- **Control flow** allows your program to make decisions with `if`, `elif`, and `else`.
- **Loops** (`for`, `while`) repeat code, enabling programs to process lists, handle repeated input, or perform calculations.
- **Special statements** like `break` and `continue` give you extra control over loop execution.
- When you combine these features, you can create powerful, flexible programs that interact with users and handle a wide variety of tasks.

Typical Interview/Quiz Questions:

- What's the difference between `=` and `==`?
- How do `for` and `while` loops differ?
- What does the `break` statement do in a loop?
- How can you check if a string starts with a vowel?
- What is the output of:

```
for i in range(1, 5):
    if i % 2 == 0:
        continue
    print(i)
```

Q&A

- Ask questions, clarify doubts, and discuss real-world uses of control flow and loops.

B. Homework Assignment: Fibonacci Sequence

What is the Fibonacci Sequence? The **Fibonacci sequence** is a famous mathematical series where **each number after the first two is the sum of the previous two numbers**.

- It starts with 0 and 1.
- The sequence goes: `0, 1, 1, 2, 3, 5, 8, 13, 21, ...`
- **Mathematical definition:** For `n  0`,

Latex syntax

$$F(0) = 0, \quad F(1) = 1F(n) = F(n-1) + F(n-2), \quad \text{for } n \geq 2$$

The Fibonacci sequence appears in nature (like the arrangement of leaves or the spiral patterns of shells), computer science, and many other areas.

Homework Instructions Write a Python program that:

1. **Asks the user how many terms** of the Fibonacci sequence to generate.
2. **Generates and prints** the Fibonacci sequence up to that many terms.
3. **Displays only the even Fibonacci numbers** from the sequence on a separate line.
4. **Calculates and prints the sum** of all Fibonacci numbers generated.
5. **Handles invalid input** (like zero or negative numbers) gracefully.

Bonus challenge:

- After finishing, allow the user to repeat the process for a new number of terms.

Example Output:

```
How many terms? 8

Fibonacci sequence:
0 1 1 2 3 5 8 13

Even Fibonacci numbers:
0 2 8

Sum of all Fibonacci numbers: 34
```

Starter Code (Optional):

```python
n = int(input("How many terms? "))

a, b = 0, 1
fib_sum = 0

print("Fibonacci sequence:")
for i in range(n):
    print(a, end=' ')
    fib_sum += a
    a, b = b, a + b

# Students: Add code to collect and display even numbers, and show the sum at the end.
```

Tip: Try breaking the problem into steps:

- Use a loop to generate Fibonacci numbers,
- Use an `if` statement to check for even numbers,
- Use a variable to keep track of the sum.

Bring your solution and questions to the next session for discussion and feedback!

Keep coding!

Lesson 03

Exercise

LESSON 3: PYTHON CONTROL FLOW (Exercises)

Hands-On Practice of Lesson 1 and Lesson 2

---

1. Even or Odd?

Write a program that asks the user for a number and prints whether it is even or odd.

---

2. Positive, Negative, or Zero

Prompt the user to enter a number. Print whether the number is positive, negative, or zero.

---

4. Grade Evaluator

Ask the user to enter a test score (0-100). Print out the corresponding grade:

- 90-100: A
- 80-89: B
- 70-79: C
- 60-69: D
- Below 60: F

---

5. Multiplication Table

Write a program that prints the multiplication table for a number provided by the user, from 1 to 10.

---

6. Sum of N Numbers

Ask the user for a positive integer N. Then, use a loop to calculate and print the sum of all numbers from 1 to N.

---

7. Countdown

Write a program that counts down from 10 to 1 using a `while` loop, printing each number.

---

8. Print All Even Numbers Between 1 and 50

Use a `for` loop to print all even numbers between 1 and 50.

9A. Number Guessing Game (Basic)

Generate a secret number (e.g., 7). Ask the user to guess the number until they get it right. After each guess, give a hint whether their guess was too high or too low.

9B. Number Guessing Game (Advanced)

Same as 9A: Generate a secret number (e.g., 7). Ask the user to guess the number until they get it right. After each guess, give a hint whether their guess was too high or too low.

Addition: Store all user guesses. Warns if a number is guessed again (does not count repeated guesses as new tries). And calculate the success ratio at the end. Success ratio = valid unique guesses divided by total number of tries, expresses as a percentage.

E.g.:

- 1. 100% if correct in first try,
- 2. 50% if got it after 2 tries but only 1 was valid,
- 3. 33% if got it after 3 tries
- 4. 25% …etc.

---

10. Sum of Even Numbers in a List

Given a list of numbers, write code to calculate and print the sum of all even numbers in the list.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

---

11. Find the Largest Number

Write a program that takes 5 numbers from the user (one at a time) and prints the largest one.

---

12. Simple ATM Menu

Simulate an ATM menu:

- 1. Check balance
- 2. Deposit
- 3. Withdraw
- 4. Exit

Use `if-elif-else` statements and a loop to allow the user to perform multiple actions until they choose to exit.

---

13. ** AbraKadabra**

Write a program that prints numbers from 1 to 50.

- For multiples of 3, print “Abra” instead of the number.
- For multiples of 5, print “Kadabra”.
- For multiples of both 3 and 5, print “AbraKadabra”.

---

14. Palindrome Checker

Ask the user for a word. Print whether it is a palindrome (the same forwards and backwards).

Lesson 3- Python Control Flow (Solutions)

LESSON 3: PYTHON CONTROL FLOW (Exercises & Solutions)

Instructions

- Try to solve each problem on your own before checking the solution.
- If you get stuck, look at the hint, then try again before looking at the code.

---

1. Even or Odd?

Task: Ask the user for a number and print whether it is even or odd.

Solution:

```
num = int(input("Enter a number: "))
if num % 2 == 0:
    print("Even")
else:
    print("Odd")
```

---

2. Positive, Negative, or Zero

Task: Prompt the user to enter a number. Print whether the number is positive, negative, or zero.

Solution:

```
num = float(input("Enter a number: "))
if num > 0:
    print("Positive")
elif num < 0:
    print("Negative")
else:
    print("Zero")
```

---

4. Grade Evaluator

Task: Get a score from the user (0-100). Print the corresponding letter grade.

Solution:

```python
score = int(input("Enter your score (0-100): "))
if score >= 90:
    print("A")
elif score >= 80:
    print("B")
elif score >= 70:
    print("C")
elif score >= 60:
    print("D")
else:
    print("F")
```

---

5. Multiplication Table

Task: Print the multiplication table for a number (1 to 10).

Solution:

```python
num = int(input("Enter a number: "))
for i in range(1, 11):
    print(f"{num} x {i} = {num * i}")
```

---

6. Sum of N Numbers

Task: Calculate the sum from 1 to N.

Solution:

```python
n = int(input("Enter a positive integer: "))
total = 0
for i in range(1, n + 1):
    total += i
print("Sum:", total)
```

---

7. Countdown

Task: Count down from 10 to 1 using a `while` loop.

Solution:

```python
i = 10
while i > 0:
    print(i)
    i -= 1
    #i = i - 1
```

---

8. Print All Even Numbers Between 1 and 50

Task: Print all even numbers between 1 and 50.

Solution:

```
for i in range(1, 51):
    if i % 2 == 0:
        print(i)
```

9A. Number Guessing Game (Basic)

Task: Guess a secret number (set to 7).

Solution:

```
secret = 7
guess = None
while guess != secret:
    guess = int(input("Guess the number: "))
    if guess < secret:
        print("Too low!")
    elif guess > secret:
        print("Too high!")
    else:
        print("Correct!")
```

9B. Number Guessing Game (Advanced)r

Task: Guess a secret number (set to 7). Calculates the success ratio and warns user of repeated guess

```
print("\n"")
print("Welcome to the Guessing Game!\n")
print("Try to guess the secret number between 1 and 10.")


secret = 7
guesses = set()  # Stores unique valid guesses
all_tries = 0    # Counts every input (including repeats)

while True:
    guess = int(input("Guess the number: "))
    all_tries += 1
    if guess in guesses:
        print("You already tried this number! Try a different one.")
        all_tries -= 1
        continue  # Don't count this as a new valid guess
```

```
    guesses.add(guess)
    if guess < secret:
        print("Too low!")
    elif guess > secret:
        print("Too high!")
    else:
        print("Correct!")
        break

print("\nYour guesses:", sorted(guesses))
valid_guesses = len(guesses)
if all_tries > 0:
    success_ratio3 = ( 10 / valid_guesses ) * 10
    print(f"Success ratio: {success_ratio3:.1f}%")
```

10. Sum of Even Numbers in a List

Task: Sum all even numbers in a list.

Solution:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
total = 0
for n in numbers:
    if n % 2 == 0:
        total += n
print("Sum of even numbers:", total)
```

---

11. Find the Largest Number

Task: Get 5 numbers from the user and print the largest.

Solution:

```
largest = None
for i in range(5):
    num = float(input("Enter a number: "))
    if (largest is None) or (num > largest):
        largest = num
print("The largest number is:", largest)
```

---

12. Simple ATM Menu

Task: Simulate a simple ATM menu using loops and control flow.

Solution:

```
balance = 1000
while True:
```

```
    print("\n1. Check balance\n2. Deposit\n3. Withdraw\n4. Exit")
    choice = input("Choose an option: ")
    if choice == "1":
        print("Balance:", balance)
    elif choice == "2":
        amount = float(input("Enter amount to deposit: "))
        balance += amount
        print("Deposited:", amount)
    elif choice == "3":
        amount = float(input("Enter amount to withdraw: "))
        if amount <= balance:
            balance -= amount
            print("Withdrawn:", amount)
        else:
            print("Insufficient funds!")
    elif choice == "4":
        print("Goodbye!")
        break
    else:
        print("Invalid choice.")
```

---

13. AbraKadabra

Task: Print numbers from 1 to 50, replace multiples of 3 with "Abra", 5 with "Kadabra", both with "AbraKadabra".

Solution:

```
for i in range(1, 51):
    if i % 3 == 0 and i % 5 == 0:
        print("AbraKadabra")
    elif i % 3 == 0:
        print("Abra")
    elif i % 5 == 0:
        print("Kadabra")
    else:
        print(i)
```

---

14. Palindrome Checker (advanced)

Task: Ask the user for a word. Print whether it is a palindrome.

Solution:

```
word = input("Enter a word: ")
if word == word[::-1]:
    print("Palindrome")
```

```
else:
    print("Not a palindrome")
```

or

```
word = input("Enter a word: ")
is_palindrome = True

for i in range(len(word) // 2):
    if word[i] != word[-(i + 1)]:
        is_palindrome = False
        break

if is_palindrome:
    print("Palindrome")
else:
    print("Not a palindrome")
```

Keep Coding!

Lesson 04

Ii. Introduction To Lists And Tuples

LESSON 4: PYTHON DATA STRUCTURES - PART 1 (LISTS AND TUPLES)

II. Introduction to Lists and Tuples (15–20 min)

A. Why Do We Need Collections?

- In many programs, we need to store multiple values—not just one.
  - Example: A list of student names, the temperatures for each day of a week, all items in a shopping cart.
- Instead of using many variables (`name1`, `name2`, `name3`…), we use a **collection**.
- Python provides several types of collections; today, we focus on **lists** and **tuples**.

B. What is a List?

- **Definition**: A list is a collection of values/items stored in a single variable. Lists can hold items of any type—integers, strings, floats, even other lists.
- **Properties**:
  - **Ordered**: Items have a defined order. Each item has an index (starting from 0).
  - **Mutable (changeable)**: You can change, add, or remove items after the list is created.
  - **Allows duplicates**: The same value can appear more than once.
  - **Syntax**: Square brackets `[ ]`.

Example:

```
# Creating a list of numbers
numbers = [10, 20, 30, 40]

# Creating a list of strings
fruits = ['apple', 'banana', 'cherry']

# A list can have mixed types
mixed = [1, "hello", 3.14, True]
```

- **Use cases**:
  - A student's test scores throughout the semester.
  - All user inputs in a game.
  - Shopping cart items in an online store.

C. What is a Tuple?

- **Definition**: A tuple is a collection **very similar to a list**, but **immutable**—once created, its items cannot be changed, added, or removed.
- **Properties**:
  - **Ordered**: Like lists, each item has a fixed position (index).
  - **Immutable (unchangeable)**: You cannot modify the tuple after it is created.
  - **Allows duplicates**: Repeating values are fine.
  - **Syntax**: Parentheses `( )`.

Declaring Tuples:

```
# Creating a tuple of numbers
coordinates = (3, 5)

# Tuple of strings
colors = ('red', 'green', 'blue')

# Single-value tuple (note the comma!)
one_value = (42,)

# Tuple without parentheses (not recommended, but possible)
data = 1, 2, 3  # This is a tuple!
```

Immutability Example:

```
person = ('Alice', 30)
# person[0] = 'Bob'  # This will cause an error!
```

D. When to Use Lists vs Tuples?

Feature	List	Tuple
Syntax	`[1, 2, 3]`	`(1, 2, 3)`
Mutable	Yes (can change items)	No (cannot change items)
Use Cases	Data that changes (cart)	Fixed data (coordinates, days)
Speed	Slightly slower	Slightly faster (read-only)
Methods	Many (append, remove, ...)	Few (mainly count, index)

Examples:

- **Use a List when:**
  - You need to add/remove/change items
  - Example: Keeping track of scores as a game progresses
- **Use a Tuple when:**
  - The data must not change (protect it by making it immutable)
  - Example: (x, y) coordinates of a point, dates, RGB color values

---

E. Practical Examples and Why Tuples are Useful

1. **Coordinates Example (Tuples):**
   - Coordinates in 2D/3D space should not change after they're set.

```
point = (10, 20)
# x, y = point  # tuple unpacking
```

2. **Function Returning Multiple Values (Tuple Packing/Unpacking):**

```
def min_max(nums):
    return (min(nums), max(nums))

result = min_max([2, 8, 3, 5])
print("Min:", result[0], "Max:", result[1])

# Tuple unpacking
min_val, max_val = min_max([2, 8, 3, 5])
print("Min:", min_val, "Max:", max_val)
```

3. **Days of the Week (Tuple):**
   - Days never change, so we use a tuple.

```
days = ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday')
```

4. **List Example (Changeable Data):**
   - Adding items to a shopping cart:

```
cart = ['apple', 'banana']
cart.append('orange')
cart[0] = 'grape'
```

F. Summary Table

Property	List	Tuple
Syntax	`[ ]`	`( )`
Ordered	Yes	Yes
Mutable	Yes	No
Methods	Many	Few
Use case	Changeable	Fixed
Example	Scores list	Coordinates

G. Discussion:

- Why do you think Python has both lists and tuples?
- What would happen if you tried to change a tuple?

Recap

Use **lists** when you need to work with changeable sequences of items. Use **tuples** when you want to *protect* data from changes, or when returning multiple values from a function.

Iii. List Operations

III. List Operations

---

1. Indexing & Slicing

A. Indexing

- **What is Indexing?**
  - Each item in a list has a position, called an *index*. Indexes start at 0 (zero-based).
  - You can use positive or negative numbers.
    - **Positive:** Start from left (`0, 1, 2, ...`)
    - **Negative:** Start from right (`-1, -2, ...`)

Example:

```
fruits = ['apple', 'banana', 'cherry', 'date']

print(fruits[0])    # 'apple'  (first element)
print(fruits[2])    # 'cherry' (third element)
print(fruits[-1])   # 'date'   (last element)
print(fruits[-2])   # 'cherry' (second last)
```

- **Usage Demo:**
  - Ask students to create a list of their favorite colors, then print the first and last color.

B. Slicing

- **What is Slicing?**
  - Slicing lets you extract a *portion* (sublist) from a list using `list[start:stop]`.
  - The slice includes the `start` index, but **not** the `stop` index (`stop` is excluded).

Syntax:

```
list[start:stop]
```

Examples:

```
numbers = [0, 1, 2, 3, 4, 5, 6]

print(numbers[1:4])   # [1, 2, 3]
print(numbers[:3])    # [0, 1, 2]
print(numbers[3:])    # [3, 4, 5, 6]
print(numbers[-3:])   # [4, 5, 6] (last three items)
```

- **Usage Demo:**
  - Given a list of 7 days, print the weekdays only (slice out the first 5 days).

---

2. Modifying Lists

A. Changing Values

- **Modify an existing element by index:**

```
fruits = ['apple', 'banana', 'cherry']
fruits[1] = 'blueberry'
print(fruits)   # ['apple', 'blueberry', 'cherry']
```

- **Demo:**
  - Students change their favorite animal in a list from "cat" to "lion".

B. Adding Elements

- **append():** Adds to the end.
- **insert():** Adds at a specific position.

```
fruits = ['apple', 'banana']
fruits.append('orange')
print(fruits)   # ['apple', 'banana', 'orange']

fruits.insert(1, 'kiwi')  # Insert at index 1
print(fruits)   # ['apple', 'kiwi', 'banana', 'orange']
```

- **Demo:**
  - Students add a new favorite movie to their list.

C. Removing Elements

- **remove(value):** Removes first occurrence of value.
- **del list[index]:** Deletes by index.
- **pop():** Removes and returns the last item (can take an index).

```
fruits = ['apple', 'banana', 'cherry', 'banana']

fruits.remove('banana')  # Removes the first 'banana'
print(fruits)   # ['apple', 'cherry', 'banana']

del fruits[1]   # Deletes at index 1 ('cherry')
print(fruits)   # ['apple', 'banana']

last = fruits.pop()  # Removes last element and returns it
print(last)     # 'banana'
print(fruits)   # ['apple']
```

- **Demo:**
  - Students remove a disliked subject from their list of subjects.

D. Other Useful Operations

- **Length of a list:**

```
colors = ['red', 'blue', 'green']
print(len(colors))   # 3
```

- **Check if an item exists:**

```
if 'blue' in colors:
    print('Blue is in the list!')
```

- **Concatenate two lists:**

```
list1 = [1, 2, 3]
list2 = [4, 5]
combined = list1 + list2
print(combined)      # [1, 2, 3, 4, 5]
```

- **Repeat a list:**

```
repeat = [0] * 5
print(repeat)        # [0, 0, 0, 0, 0]
```

---

3. Iterating Over Lists

A. For Loop (Most Common Way)

- **Go through each element:**

```
fruits = ['apple', 'banana', 'cherry']
for fruit in fruits:
    print(fruit)
```

- **Demo:**
  - Ask students to print every item in their "to-do" list.

B. While Loop (Alternative)

- **Using indexes to loop:**

```
i = 0
while i < len(fruits):
    print(fruits[i])
    i += 1
```

- **When to use?**
  - When you need access to the index, or want more control over the loop.

C. Looping with Indexes (Enumerate)

- **Get both index and value:**

```
for i, fruit in enumerate(fruits):
    print(f"Index {i}: {fruit}")
```

Student Demos & Mini-Exercises

Try these in class:

1. Create a list of 5 countries. Print the third one.
2. Add a new country, then remove the first one.
3. Slice and print the last two countries.
4. Loop through your list and print each country in uppercase.

5. Concatenate your list with a friend's list.

Summary Table

Operation	Example Code	Output / Explanation
Indexing	`fruits[1]`	'banana' (second item)
Slicing	`fruits[1:3]`	['banana', 'cherry']
Change value	`fruits[1] = 'kiwi'`	Replaces 'banana' with 'kiwi'
Append	`fruits.append('orange')`	Adds 'orange' at the end
Insert	`fruits.insert(0, 'pear')`	Adds 'pear' at start
Remove value	`fruits.remove('apple')`	Removes first 'apple'
Delete index	`del fruits[0]`	Removes item at index 0
Pop	`fruits.pop()`	Removes & returns last item
Check in list	`'apple' in fruits`	True if 'apple' exists
Length	`len(fruits)`	Number of items in list
Concatenate	`list1 + list2`	Joins two lists

Iv. Tuple Operations

IV. Tuple Operators and Operations

A. Tuple Operators

1. Concatenation (+)

- You can join two or more tuples using the `+` operator.
- The result is a new tuple containing the elements of both.

```
a = (1, 2, 3)
b = (4, 5)
c = a + b
print(c)  # Output: (1, 2, 3, 4, 5)
```

2. Repetition (*)

- You can repeat a tuple multiple times using the `*` operator.

```
numbers = (0, 1)
result = numbers * 3
print(result)  # Output: (0, 1, 0, 1, 0, 1)
```

3. Membership (`in`, `not in`)

- Check if an item is in a tuple with `in`, or not in a tuple with `not in`.

```
colors = ('red', 'green', 'blue')
print('green' in colors)      # True
print('yellow' not in colors) # True
```

4. Comparison Operators

- Tuples can be compared using `<`, `<=`, `>`, `>=`, `==`, `!=`.
- Python compares element by element (left to right):

```
a = (1, 2, 3)
b = (1, 2, 4)
print(a < b)   # True, because 3 < 4
```

B. Tuple Operations (Methods)

Tuples have only two built-in methods because they are immutable:

1. `count(x)`

- Returns the number of times `x` appears in the tuple.

```
t = (1, 2, 2, 3, 2)
print(t.count(2))  # Output: 3
```

2. `index(x)`

- Returns the first index at which `x` appears.
- Raises a `ValueError` if `x` is not found.

```
t = ('a', 'b', 'c', 'b')
print(t.index('b'))  # Output: 1
# print(t.index('z'))  # ValueError: tuple.index(x): x not in tuple
```

C. Built-in Functions Useful with Tuples

- **`len()`** — Number of elements

  ```
  tup = (1, 2, 3, 4)
  print(len(tup))  # 4
  ```

- **`min()` / `max()`** — Smallest / largest item

  ```
  tup = (10, 5, 20)
  print(min(tup))  # 5
  print(max(tup))  # 20
  ```

- **`sum()`** — Sum all elements (only for numbers)

  ```
  tup = (1, 2, 3)
  print(sum(tup))  # 6
  ```

- **`sorted()`** — Returns a sorted **list** (does not change the tuple!)

```
tup = (3, 1, 2)
sorted_tup = sorted(tup)
print(sorted_tup)  # [1, 2, 3]
```

- **`tuple()`** — Converts a list (or any iterable) to a tuple

```
numbers = [1, 2, 3]
t = tuple(numbers)
print(t)  # (1, 2, 3)
```

D. Iterating Over Tuples

- **For Loop:**

```
fruits = ('apple', 'banana', 'cherry')
for fruit in fruits:
    print(fruit)
```

- **With Index:**

```
for i in range(len(fruits)):
    print(f"Index {i}: {fruits[i]}")
```

- **Unpacking in Loops (useful for tuples of pairs):**

```
points = ((1, 2), (3, 4), (5, 6))
for x, y in points:
    print(f"x={x}, y={y}")
```

E. Tuple Packing and Unpacking

- **Packing:** Grouping values into a tuple, often done implicitly:

```
t = 1, 2, 3  # Packing
```

- **Unpacking:**

```
a, b, c = t
print(a, b, c)  # 1 2 3
```

F. Slicing Tuples

- Just like lists, you can use slicing:

```
tup = (0, 1, 2, 3, 4, 5)
print(tup[2:5])  # (2, 3, 4)
print(tup[:3])   # (0, 1, 2)
print(tup[-2:])  # (4, 5)
```

G. Practical Demos

Demo 1: Create two tuples, concatenate and repeat them.

```
a = ('x', 'y')
b = ('z',)
combo = a + b
print(combo * 2)   # ('x', 'y', 'z', 'x', 'y', 'z')
```

Demo 2: Check for existence of number 2 in tuple: (1, 2, 2, 3, 4, 2) and count occurrences of number 2.

```
nums = (1, 2, 2, 3, 4, 2)
print(2 in nums)          # True
print(nums.count(2))      # 3
print(nums.index(3))      # 3
```

Demo 3: Create a tuple and sort it (returns a list).

```
t = (5, 2, 7, 1)
print(sorted(t))          # [1, 2, 5, 7]
```

Demo 4: Tuple unpacking with function return.

```
def stats(values):
    return (min(values), max(values), sum(values))

min_v, max_v, total = stats((4, 7, 2, 9))
print(min_v, max_v, total)  # 2 9 22
```

H. Quick Practice Questions

1. Create a tuple of 5 colors. Print the first three using slicing.
2. Check if 'blue' is in your tuple.
3. Count how many times 'red' appears.
4. Concatenate your color tuple with another tuple of 2 colors.
5. Use unpacking to assign the first two colors to variables and print them.

Keep Coding!

V. Hands-On Practice

V. Hands-On Practice (20 min)

Let students try these tasks **step by step**, encourage them to experiment, and discuss their results. You can guide, check, and answer questions as they go!

---

1. Lists Practice

a. Create a list of 5 favorite movies. This shows how to create and store multiple items in a list.

```
movies = ['Inception', 'Before the rain', 'The Matrix', 'Splav Meduze', 'Top Gun']
```

---

b. Print the first and last movie. Practices indexing (positive and negative).

```
print("First movie:", movies[0])
print("Last movie:", movies[-1])
```

Expected Output:

```
First movie: Inception
Last movie: Top Gun
```

---

c. Replace the third movie. Shows how to change an item in a list (lists are mutable).

```
movies[2] = 'Interstellar'  # Change 'The Matrix' to 'Interstellar'
print(movies)
```

Expected Output:

```
['Inception', 'Before the rain', 'Interstellar', 'Splav Meduze', 'Top Gun']
```

---

d. Add a new movie at the end. *Why?* Demonstrates adding with `.append()`.

```
movies.append('The Godfather')
print(movies)
```

Expected Output:

```
['Inception', 'Before the rain', 'Interstellar', 'Splav Meduze', 'Top Gun', 'The Godfather']
```

---

e. Remove one movie by name. *Why?* Shows how to remove an item by value.

```
movies.remove('Before the rain')
print(movies)
```

Expected Output:

```
['Inception', 'Interstellar', 'Splav Meduze', 'Top Gun', 'The Godfather']
```

---

2. Tuples Practice

a. Create a tuple with 3 cities. *Why?* Demonstrates how to store multiple values in a tuple.

```
cities = ('Seoul', 'Busan', 'Pohang')
```

---

b. Print each city using a loop. Shows how to iterate over a tuple.

```
for city in cities:
    print(city)
```

Expected Output:

```
Seoul
Busan
Pohang
```

---

c. Try to change one city (observe the error). Shows immutability—tuples can't be changed.

```
# Try this (it will cause an error!)
# cities[1] = 'Asan-si'
```

Expected Output:

```
TypeError: 'tuple' object does not support item assignment
```

Explanation: Unlike lists, you **cannot** change the value of any item in a tuple after creation.

---

d. Unpack the tuple into variables. Shows how to assign tuple elements to multiple variables at once.

```
city1, city2, city3 = cities
print(city1)
print(city2)
print(city3)
```

Expected Output:

```
Seoul
Busan
Pohang
```

Extra Challenge (if time allows):

- **List:** Ask the user for a new movie to add to their list (use `input()`), then print the updated list.
- **Tuple:** Create a tuple of (latitude, longitude) for a city and print both using unpacking.

Vi. Homework Assignment

VI. Homework Assignment

Assignment Instructions

Write a Python program that:

1. **Takes a list of numbers as input from the user.**
   - The numbers should be entered as a single line, separated by commas (e.g., `3,18,9,25,4,12`).
2. **Converts the input into a list of integers.**
3. **Sorts the list in ascending order.**
4. **Filters out all numbers less than 10.**
5. **Prints the sorted and filtered list.**

**** Some hints****

- Use `input()` to get the user's numbers as a string.
- Use `.split(",")` to break the string into separate pieces.
- Use a list comprehension to convert each piece to an integer: `[int(x) for x in list]`.
- Use the `.sort()` method or the `sorted()` function.
- Use another list comprehension to filter the numbers (`if n >= 10`).

Sample Solution (with Explanations and Comments)

```
# Step 1: Ask the user to input numbers, comma-separated
numbers_str = input("Enter numbers separated by commas: ")

# Step 2: Split the string into a list of strings
numbers_list_str = numbers_str.split(",")

# Step 3: Convert each string to an integer using list comprehension
numbers = [int(x.strip()) for x in numbers_list_str]

# Step 4: Sort the list in ascending order
numbers.sort()  # Or: numbers = sorted(numbers)

# Step 5: Filter out numbers less than 10 (keep only >= 10)
filtered_numbers = [n for n in numbers if n >= 10]

# Step 6: Print the final sorted and filtered list
print("Sorted and filtered list:", filtered_numbers)
```

Lesson 05

Python Data Structures - Part 1-Lab

Lesson 5: Python Data Structures - Part 1

---

1. Tuple Creation & Access

Create a tuple called `fruits` with 5 fruit names. Print the first and last fruit using indexing.

---

2. Unpacking Tuples

Create a tuple `(name, age, country)`. Unpack it into separate variables and print them in a formatted string: `"Name: ___, Age: ___, Country: ___"`.

---

3. Tuple Immutability

Try to change the second item of a tuple. Observe and print the error message.

---

4. Tuple with Mixed Data Types

Create a tuple with an integer, a float, a string, and a boolean. Print each item and its data type.

---

5. List Creation and Manipulation

Create a list `numbers` with values [10, 20, 30, 40, 50].

- Add 60 to the end.
- Insert 25 at position 1.
- Remove 40 from the list.
- Print the final list.

---

6. Slicing Lists and Tuples

Given a list `nums = [5, 10, 15, 20, 25, 30]`, print:

- The first three numbers.
- The last two numbers.

---

7. List Methods Practice

Given a list of cities, practice these methods:

- `append()`, `pop()`, `reverse()`, `sort()`
- Print the list after each operation.

---

8. Converting Between Tuples and Lists

Convert a tuple `(1, 2, 3, 4)` to a list, add `5`, then convert it back to a tuple. Print the final tuple.

---

9. Looping Through Tuples and Lists

Write a function that takes a list of integers and prints only the even numbers.

---

10. Nested Lists

Create a 2D list (matrix) for 3 students' marks in 3 subjects each. Print each student's marks in a readable format.

---

11. Using Lists with Input

Ask the user to enter 5 numbers (using `input()` in a loop). Store them in a list and print the maximum and minimum value.

---

12. List Comprehensions

Given a list of numbers, create a new list containing only their squares using a list comprehension.

---

13. Finding Duplicates in a List

Given a list with some repeated numbers, print all numbers that appear more than once.

---

14. Functions Returning Tuples

Write a function that receives two numbers and returns both their sum and product as a tuple. Call the function and print the results.

---

15. Combining Strings and Lists

Given a string of comma-separated values, split it into a list. Then, sort and join the list back into a comma-separated string and print it.

Solution-Python Data Structures - Part 1-Lab

Solutions: Python Data Structures - Tuples and Lists

1. Tuple Creation & Access

```
fruits = ("apple", "banana", "orange", "grape", "mango")
print(fruits[0])   # First fruit: apple
print(fruits[-1])  # Last fruit: mango
```

---

2. Unpacking Tuples

```
person = ("Alice", 21, "USA")
name, age, country = person
print(f"Name: {name}, Age: {age}, Country: {country}")
```

---

3. Tuple Immutability

```
my_tuple = (1, 2, 3)
try:
    my_tuple[1] = 20
except TypeError as e:
    print("Error:", e)
# Output: Error: 'tuple' object does not support item assignment
```

---

4. Tuple with Mixed Data Types

```
data = (10, 3.14, "hello", True)
for item in data:
    print(item, type(item))
```

---

5. List Creation and Manipulation

```
numbers = [10, 20, 30, 40, 50]
numbers.append(60)
numbers.insert(1, 25)
numbers.remove(40)
print(numbers)  # Output: [10, 25, 20, 30, 50, 60]
```

---

6. Slicing Lists and Tuples

```
nums = [5, 10, 15, 20, 25, 30]
print(nums[:3])   # [5, 10, 15]
print(nums[-2:])  # [25, 30]
```

---

7. List Methods Practice

```
cities = ["London", "Paris", "Rome"]
cities.append("Berlin")
print(cities)   # ['London', 'Paris', 'Rome', 'Berlin']

cities.pop()
print(cities)   # ['London', 'Paris', 'Rome']

cities.reverse()
print(cities)   # ['Rome', 'Paris', 'London']

cities.sort()
print(cities)   # ['London', 'Paris', 'Rome']
```

---

8. Converting Between Tuples and Lists

```
my_tuple = (1, 2, 3, 4)
temp_list = list(my_tuple)
temp_list.append(5)
my_tuple = tuple(temp_list)
print(my_tuple)  # (1, 2, 3, 4, 5)
```

---

9. Looping Through Tuples and Lists

```
def print_even_numbers(numbers):
    for n in numbers:
        if n % 2 == 0:
            print(n)

print_even_numbers([1, 2, 3, 4, 5, 6])
# Output: 2 4 6
```

---

10. Nested Lists

```
marks = [
    [85, 90, 92],    # Student 1
    [78, 88, 80],    # Student 2
```

```
    [90, 91, 89]      # Student 3
]
for i, student_marks in enumerate(marks, start=1):
    print(f"Student {i}: {student_marks}")
```

---

11. Using Lists with Input

```
numbers = []
for i in range(5):
    num = int(input("Enter a number: "))
    numbers.append(num)
print("Maximum:", max(numbers))
print("Minimum:", min(numbers))
```

---

12. List Comprehensions

```
numbers = [2, 4, 6, 8]
squares = [n ** 2 for n in numbers]
print(squares)  # [4, 16, 36, 64]
```

---

13. Finding Duplicates in a List

```
nums = [1, 2, 3, 2, 4, 5, 3, 6, 1]
duplicates = []
for n in nums:
    if nums.count(n) > 1 and n not in duplicates:
        duplicates.append(n)
print(duplicates)  # [1, 2, 3]
```

---

14. Functions Returning Tuples

```
def sum_and_product(a, b):
    return (a + b, a * b)

result = sum_and_product(5, 3)
print("Sum:", result[0])
print("Product:", result[1])
```

---

15. Combining Strings and Lists

```
csv = "banana,apple,mango,pear"
fruits_list = csv.split(",")
```

```
fruits_list.sort()
sorted_csv = ",".join(fruits_list)
print(sorted_csv)  # apple,banana,mango,pear
```

Lesson 06

Ii. Introduction To Dictionaries And Sets

LESSON 6: PYTHON DATA STRUCTURES - PART 2 (DICTIONARIES AND SETS)

II. Introduction to Dictionaries and Sets

What is a Dictionary?

A **dictionary** in Python is an **unordered**, **mutable** collection of key-value pairs. Each key in a dictionary must be unique and immutable (like strings, numbers, or tuples), and is used to store and retrieve values efficiently.

Syntax:

```
my_dict = {'name': 'Alice', 'age': 25}
```

- Here, `'name'` and `'age'` are **keys**.
- `'Alice'` and 25 are their corresponding **values**.

Accessing Values: You can access a value by referencing its key:

```
print(my_dict['name'])   # Output: Alice
```

Adding or Updating Entries:

```
my_dict['email'] = 'alice@example.com'  # Adds a new key-value pair
my_dict['age'] = 26                     # Updates existing key
```

Removing Entries:

```
del my_dict['age']                      # Removes the key 'age'
```

Useful Methods:

- `my_dict.keys()` returns all keys.
- `my_dict.values()` returns all values.
- `my_dict.items()` returns all key-value pairs.

Example Scenario:

- **Student Grades:** Store student names as keys and their grades as values.

  ```
  grades = {'Alice': 95, 'Bob': 88, 'Charlie': 72}
  print(grades['Bob'])  # Output: 88
  ```

- **Phone Book:** Map names to phone numbers.

```
phone_book = {'John': '555-1234', 'Jane': '555-5678'}
```

- **Counting Occurrences:** Count how many times each word appears in a sentence.

```
# This is what it is
word_count = {'This': 1, 'is': 2, 'what': 1, 'it': 1}
```

What is a Set?

A **set** is an **unordered** collection of **unique** elements. Sets are mutable, but elements must be immutable.

Syntax:

```
my_set = {1, 2, 3}
```

- Duplicates are automatically removed.

```
s = {1, 2, 2, 3}
print(s)  # Output: {1, 2, 3}
```

Adding Elements:

```
my_set.add(4)    # Adds 4 to the set
```

Removing Elements:

```
my_set.remove(2) # Removes 2 from the set (raises error if 2 not present)
my_set.discard(5) # Removes 5 if present, does nothing if not
```

Useful Methods:

- `my_set.union(other_set)` combines two sets.
- `my_set.intersection(other_set)` finds common elements.
- `my_set.difference(other_set)` finds elements in one set but not the other.

Example Scenario:

- **Removing Duplicates:** Turn a list into a set to remove duplicates.

```
numbers = [1, 2, 2, 3, 4, 4]
unique_numbers = set(numbers)
print(unique_numbers)  # Output: {1, 2, 3, 4}
```

- **Membership Testing:** Quickly check if an item exists.

```
vowels = {'a', 'e', 'i', 'o', 'u'}
print('e' in vowels)  # Output: True
```

- **Finding Shared Items:** Find common students in two classes.

```
class1 = {'Alice', 'Bob', 'Charlie'}
class2 = {'Bob', 'David', 'Ella'}
print(class1.intersection(class2))  # Output: {'Bob'}
```

Why use them?

- **Dictionaries:** Use dictionaries when you want to associate keys with values, such as mapping student names to their grades, usernames to passwords, or product IDs to product details. They allow for very fast lookups, additions, and deletions based on the key.
- **Sets:** Use sets for tasks that involve uniqueness and membership tests, such as removing duplicates from a list, checking if a value exists, or finding common or unique items between two groups.

Summary Table:

Structure	Order	Mutable	Unique Elements	Key-Value Pair	Main Use Case
Dictionary	Unordered	Yes	Keys are unique	Yes	Fast lookup by key, data mapping
Set	Unordered	Yes	Yes	No	Uniqueness, membership tests

Quick Challenge:

- Create a dictionary of three countries and their capitals.
- Create a set of five numbers, with at least two repeated, and print the set.

Iii.Dictionary Operations

III. Dictionary Operations

1. Creating a Dictionary A dictionary can be created using curly braces `{}` or the `dict()` constructor.

```
# Using curly braces
student = {'name': 'John', 'age': 20, 'courses': ['Math', 'CompSci']}
print(student)

# Using dict() constructor
employee = dict(name='Jane', department='HR', salary=50000)
print(employee)
```

2. Accessing Values Retrieve values using keys. If the key doesn't exist, using `[]` raises an error, but `.get()` returns `None` (or a default value if provided).

```
print(student['name'])      # Output: John
print(student.get('age'))   # Output: 20
print(student.get('grade')) # Output: None (no error)
print(student.get('grade', 'Not Assigned')) # Output: Not Assigned
```

Tip: Use `.get()` to avoid errors when a key might not exist.

3. Adding/Updating Values Add a new key-value pair or update an existing key by assignment.

```
student['phone'] = '555-1234'   # Adds a new key-value pair
student['age'] = 21             # Updates the value for 'age'
print(student)
```

Bulk update: You can update multiple values at once with `.update()`:

```
student.update({'name': 'Jonathan', 'age': 22, 'email': 'jon@example.com'})
print(student)
```

4. Removing Items Remove a key-value pair using `del`, `.pop()`, or `.popitem()`:

```
del student['age']                      # Removes 'age'
phone = student.pop('phone')            # Removes 'phone' and returns its value
print(student)

# Remove and return the last inserted key-value pair (Python 3.7+)
last_item = student.popitem()
print(last_item)
```

Clear all items:

```
student.clear()
print(student)  # Output: {}
```

5. Dictionary Methods Some useful dictionary methods and ways to work with them:

- **Keys:** Get a view of all keys

  ```
  print(student.keys())     # dict_keys(['name', 'courses'])
  ```

- **Values:** Get a view of all values

  ```
  print(student.values())   # dict_values(['John', ['Math', 'CompSci']])
  ```

- **Items:** Get a view of key-value pairs (tuples)

  ```
  print(student.items())    # dict_items([('name', 'John'), ('courses', ['Math', 'CompSci'])
  ```

Iterating through a dictionary:

```
for key in student:
    print(key, student[key])
```

```
for key, value in student.items():
    print(key, value)
```

Checking if a key exists:

```
if 'name' in student:
    print('Name found!')
```

6. Example: Counting Words A common use case for dictionaries is to count occurrences:

```
sentence = "this is a test this is only a test"
word_count = {}
for word in sentence.split():
    word_count[word] = word_count.get(word, 0) + 1
print(word_count)
# Output: {'this': 2, 'is': 2, 'a': 2, 'test': 2, 'only': 1}
```

7. More Examples and Useful Functions a. Dictionary Comprehensions

Quickly create a dictionary from a sequence:

```
# Squares of numbers 1-5
squares = {x: x*x for x in range(1, 6)}
print(squares)  # Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

b. Merging Dictionaries

Combine two dictionaries (Python 3.5+):

```
dict1 = {'a': 1, 'b': 2}
dict2 = {'b': 3, 'c': 4}
merged = {**dict1, **dict2}
print(merged)  # Output: {'a': 1, 'b': 3, 'c': 4}
```

c. Using Default Values with defaultdict

The `collections` module has a helpful `defaultdict` for auto-initializing values:

```
from collections import defaultdict

word_count = defaultdict(int)
for word in sentence.split():
    word_count[word] += 1
print(dict(word_count))
```

Real World Examples

- **Contacts/Phone Book:** Store contact names as keys, and phone numbers or info as values.

  ```
  contacts = {
      'Alice': {'phone': '123-4567', 'email': 'alice@mail.com'},
  ```

```
    'Bob': {'phone': '987-6543'}
}
```

- **Inventory Management:** Keep track of product names/IDs and their stock.

  `inventory = {'apples': 30, 'bananas': 12, 'oranges': 7}`

- **Student Grades:** Map students to their list of grades.

  `grades = {'Alice': [90, 85], 'Bob': [78, 80]}`

- **Configuration Settings:** Store application settings, such as theme, language, etc.

  `settings = {'theme': 'dark', 'language': 'en'}`

- **Counting Occurrences:** Like the word count example above, this can be used for counting anything—votes, reactions, etc.

Summary Table: Useful Dictionary Methods

Method	Description	Example
`dict.get(key)`	Returns value for key or None/default	`my_dict.get('a', 0)`
`dict.update(...)`	Updates dictionary with another dict or key-values	`my_dict.update({'a': 10})`
`dict.pop(key)`	Removes specified key and returns value	`my_dict.pop('a')`
`dict.clear()`	Removes all items	`my_dict.clear()`
`dict.keys()`	Returns view of all keys	`my_dict.keys()`
`dict.values()`	Returns view of all values	`my_dict.values()`
`dict.items()`	Returns view of all key-value pairs	`my_dict.items()`
`dict.setdefault()`	Returns value if key exists, else sets and returns new	`my_dict.setdefault('a', 100)`
`dict.fromkeys()`	Creates dict from sequence of keys with given value	`dict.fromkeys(['a', 'b'], 0)`

Quick Challenges:

1. Create a dictionary of three movies and their release years. Print all the movie names.
2. Write a program to count the number of times each letter appears in a word (hint: use a dictionary).
3. Given two dictionaries of student marks, merge them so that the second dictionary's data overwrites any duplicates.

Iv. Introduction To Sets

IV. Introduction to Sets

1. Creating Sets A **set** is an unordered collection of unique, immutable elements.

Using curly braces:

```
nums = {1, 2, 3, 4}
```

Using the set() constructor (especially for lists/strings):

```
nums2 = set([3, 4, 5, 6])
print(nums, nums2)  # Output: {1, 2, 3, 4} {3, 4, 5, 6}
```

Important:

Empty set: Use `set()`, not `{}` (which makes an empty dictionary).

```
empty_set = set()
```

2. Adding and Removing Elements Adding an element:

```
nums.add(5)
print(nums)  # Output: {1, 2, 3, 4, 5}
```

If the element already exists, nothing changes.

Removing an element:

```
nums.remove(1)   # Removes 1, raises error if 1 not found
print(nums)
nums.discard(10) # Removes 10 if present, does nothing if not (no error)
```

Remove and return an arbitrary element:

```
popped = nums.pop()
print("Removed:", popped)
```

Clear all elements:

```
nums.clear()
print(nums)  # Output: set()
```

3. Set Operations Sets support powerful operations for comparing and combining groups of items:

Union: Elements in either set

```
nums = {2, 3, 4, 5}
nums2 = {3, 4, 5, 6}
union = nums | nums2
print(union)                    # Output: {2, 3, 4, 5, 6}
# OR
print(nums.union(nums2))
```

Intersection: Elements in both sets

```
intersection = nums & nums2
print(intersection)             # Output: {3, 4, 5}
# OR
print(nums.intersection(nums2))
```

Difference: Elements in the first set but not the second

```
difference = nums - nums2
print(difference)              # Output: {2}
# OR
print(nums.difference(nums2))
```

Symmetric Difference: Elements in either set, but not both

```
sym_diff = nums ^ nums2
print(sym_diff)                # Output: {2, 6}
# OR
print(nums.symmetric_difference(nums2))
```

Membership test:

```
print(3 in nums)    # True if 3 is in nums
print(10 in nums)   # False
```

4. Example: Removing Duplicates from a List Sets make it easy to remove duplicates:

```
items = [1, 2, 2, 3, 4, 4, 5]
unique_items = set(items)
print(unique_items)  # Output: {1, 2, 3, 4, 5}
```

- If you need the result as a list again:

  ```
  unique_list = list(unique_items)
  print(unique_list)
  ```

5. Useful Set Methods

Method	Description	Example
`add(element)`	Add an element to the set	`nums.add(7)`
`remove(element)`	Remove element (error if not present)	`nums.remove(3)`
`discard(element)`	Remove element (no error if not present)	`nums.discard(10)`
`pop()`	Remove and return an arbitrary element	`nums.pop()`
`clear()`	Remove all elements	`nums.clear()`
`union(other_set)`	All unique elements from both sets	`a.union(b)`
`intersection(other_set)`	Only elements in both sets	`a.intersection(b)`
`difference(other_set)`	Elements only in the first set	`a.difference(b)`
`symmetric_difference(other)`	Elements in either, but not both	`a.symmetric_difference(b)`
`issubset(other)`	True if set is subset of another	`a.issubset(b)`
`issuperset(other)`	True if set is superset of another	`a.issuperset(b)`
`copy()`	Returns a shallow copy of the set	`a.copy()`

6. Real World Examples of Using Sets Removing Duplicates:

Get unique email addresses from a mailing list.

```
emails = ['a@mail.com', 'b@mail.com', 'a@mail.com']
unique_emails = set(emails)
print(unique_emails)
```

Membership Test:
Check if a user has admin privileges.

```
admins = {'alice', 'bob'}
username = 'charlie'
if username in admins:
  print('Admin')
else:
  print('Not admin')
```

Finding Common/Unique Items:

Find students in both math and science clubs.

```
math_club = {'Tom', 'Jane', 'Sara'}
science_club = {'Sara', 'Alex', 'Jane'}
both_clubs = math_club & science_club
print(both_clubs)  # Output: {'Jane', 'Sara'}
```

V. Hands-On Practice

V. Hands-on Practice

Below are three guided exercises to help reinforce your understanding of dictionaries and sets.

Exercise 1: Store and Retrieve Data Using a Dictionary

Goal: Practice creating a dictionary, storing user input, and retrieving values.

Instructions:

1. Create an empty dictionary called `person`.
2. Ask the user to enter their name and age.
3. Store these values in the dictionary using appropriate keys (`'name'`, `'age'`).
4. Print a personalized greeting using data from the dictionary.

Sample Solution:

```
person = {}
person['name'] = input("Enter your name: ")
person['age'] = input("Enter your age: ")
print(f"Hello {person['name']}, you are {person['age']} years old.")
```

Try this:

- Add a new key for `'city'` and ask the user for their city.
- Print a message that uses all three fields.

Exercise 2: Use a Set to Check for Duplicates

Goal: Learn how sets can help check for duplicates in a collection.

Instructions:

1. Prompt the user to enter numbers separated by spaces.
2. Convert the input string into a list of integers.
3. Convert the list to a set.
4. If the length of the set is less than the list, there were duplicates.

Sample Solution:

```
nums = [int(x) for x in input("Enter numbers separated by space: ").split()]
if len(nums) != len(set(nums)):
    print("There are duplicates!")
else:
    print("All numbers are unique.")
```

Try this:

- Print the duplicate numbers (hint: use a dictionary or a set to count).
- Let the user enter words instead of numbers and check for duplicate words.

Exercise 3: Iterating Over a Dictionary

Goal: Practice iterating through a dictionary and printing key-value pairs.

Instructions:

1. Use a dictionary of names and grades.
2. Loop through each key-value pair.
3. Print the name and corresponding grade in a formatted string.

Sample Solution:

```
grades = {'Alice': 85, 'Bob': 90, 'Charlie': 78}
for name, grade in grades.items():
    print(f"{name}: {grade}")
```

Try this:

- Calculate and print the average grade.
- Find and print the name(s) of the student(s) with the highest grade.

VI. Homework

Build a Contact Book Using Dictionaries

Goal: Create an interactive program that allows users to store, update, and search for contact information using dictionaries.

Detailed Requirements:

- Store each contact's name as the key.
- The value for each contact is another dictionary with at least `'phone'` and `'email'`.
- The program should allow the user to:
  - **Add** a new contact.
  - **Update** an existing contact.
  - **Search** for a contact by name.
  - **Exit** the program.
- Display appropriate messages for each action (e.g., "Contact added", "Contact not found").
- Use a loop to keep the program running until the user chooses to exit.

Example Starter Code:

```
contacts = {}

while True:
    print("\nContact Book")
    print("1. Add Contact")
    print("2. Update Contact")
    print("3. Search Contact")
    print("4. Exit")
    choice = input("Choose an option: ")

    if choice == '1':
        name = input("Enter name: ")
        phone = input("Enter phone: ")
        email = input("Enter email: ")
        contacts[name] = {'phone': phone, 'email': email}
        print("Contact added.")
    elif choice == '2':
        name = input("Enter name to update: ")
        if name in contacts:
            phone = input("Enter new phone: ")
            email = input("Enter new email: ")
            contacts[name] = {'phone': phone, 'email': email}
            print("Contact updated.")
        else:
            print("Contact not found.")
    elif choice == '3':
        name = input("Enter name to search: ")
        if name in contacts:
            print(f"Name: {name}")
            print(f"Phone: {contacts[name]['phone']}")
```

```
            print(f"Email: {contacts[name]['email']}")
        else:
            print("Contact not found.")
    elif choice == '4':
        break
    else:
        print("Invalid option.")
```

Tips for Success:

- Use dictionaries for flexible, fast lookups.
- Remember to check if a contact exists before updating or searching.
- Think about user experience—give clear instructions and feedback.
- Try breaking your code into functions (optional, for more advanced students).

Lesson 07

Lesson 7- Python Data Structures - Part 2 - Lab Solutions

LESSON 7: PYTHON DATA STRUCTURES - PART 2 (Exercise) - Solution

Dictionaries

---

1. Create a Contact Book

```
contacts = {
    'Alice': '123-456-7890',
    'Bob': '234-567-8901',
    'Charlie': '345-678-9012'
}
print(contacts)
```

---

2. Update a Dictionary Value

```
person = {'name': 'Tom', 'age': 30, 'city': 'New York'}
person['city'] = 'San Francisco'
print(person)
```

---

3. Delete an Entry

```
del person['age']
print(person)
```

Note: This uses the updated ***person*** *from the previous question.*

---

4. Loop Through Keys and Values

```
scores = {'Alice': 90, 'Bob': 82, 'Charlie': 95}
for name, score in scores.items():
    print(f"{name}: {score}")
```

---

5. Count Word Occurrences

```
words = ['apple', 'banana', 'apple', 'orange', 'banana', 'apple']
word_counts = {}
for word in words:
    if word in word_counts:
        word_counts[word] += 1
    else:
        word_counts[word] = 1
print(word_counts)
```

---

6. Nested Dictionaries – Student Records

```
students = {
    'Alice': {'age': 19, 'marks': 85},
    'Bob': {'age': 20, 'marks': 90},
    'Carol': {'age': 21, 'marks': 78}
}
print(students)
```

---

7. Retrieve All Keys and Values

```
sample_dict = {'x': 1, 'y': 2, 'z': 3}
print("Keys:")
for key in sample_dict.keys():
    print(key)
print("Values:")
for value in sample_dict.values():
    print(value)
```

You can replace `sample_dict` *with any dictionary.*

---

8. Dictionary from Two Lists

```
keys = ['id', 'name', 'email']
values = [1, 'Sarah', 'sarah@example.com']
combined = dict(zip(keys, values))
print(combined)
```

Sets

---

9. Remove Duplicates from a List

```
numbers = [1, 2, 2, 3, 4, 4, 5]
unique_numbers = list(set(numbers))
print(unique_numbers)
```

---

10. Set Operations – Union and Intersection

```
a = {1, 3, 5, 7}
b = {3, 4, 5, 6}
union = a | b          # or a.union(b)
intersection = a & b   # or a.intersection(b)
print("Union:", union)
print("Intersection:", intersection)
```

---

11. Check Membership

```
numbers = {2, 4, 6, 8, 10, 12}
if 10 in numbers:
    print("10 is in the set")
else:
    print("10 is not in the set")
```

---

12. Add and Remove Set Elements

```
animals = {'cat', 'dog', 'parrot'}
animals.add('hamster')
animals.remove('dog')
print(animals)
```

---

13. Common Elements in Two Lists

```
list1 = [2, 4, 6, 8, 10]
list2 = [3, 6, 9, 12, 15]
common = set(list1) & set(list2)
print("Common elements:", common)
```

Combined / Application

---

14. Unique Words in a Sentence

```
sentence = input("Enter a sentence: ")
#remove commas or stops
sentence = sentence.replace(".", " ")
words = sentence.lower().split()
unique_words = set(words)
print("Unique words:", unique_words)
```

---

15. Frequency Dictionary from a String

```
sentence = "Python Programming"
frequency = {}
for char in sentence.lower():
    if char != ' ':  # or we can replace " " with ""
        if char in frequency:
            frequency[char] += 1
        else:
            frequency[char] = 1
print(frequency)
```

16. Word Frequency Counter

```
# import string
s = "Hello hello, world today, Hello"
s = s.lower()
# Remove punctuation
s = s.translate(str.maketrans('', '', string.punctuation))
words = s.split()
freq = {}
for word in words:
    if word in freq:
        freq[word] += 1
    else:
        freq[word] = 1
print(freq)
```

16. Student Grades Analysis

```
students = {'Alice': [90, 85, 88], 'Bob': [78, 80], 'Carol': []}
averages = {}
for student, grades in students.items():
    if grades:
        avg = sum(grades) / len(grades)
        averages[student] = avg
    else:
        averages[student] = None  # or 0
print(averages)
```

Lesson 7- Python Data Structures - Part 2 - Lab

LESSON 7: PYTHON DATA STRUCTURES - PART 2 (Exercise)

Dictionaries

1. Create a Contact Book Write code to create a dictionary called `contacts` where the keys are names and the values are phone numbers. Add at least three contacts. Print the dictionary.

---

2. Update a Dictionary Value Given the dictionary below, update the city to "San Francisco":

```
person = {'name': 'Tom', 'age': 30, 'city': 'New York'}
```

---

3. Delete an Entry Remove the `'age'` key from the `person` dictionary above and print the updated dictionary.

---

4. Loop Through Keys and Values Write a loop that prints every key and value in the following dictionary:

```
scores = {'Alice': 90, 'Bob': 82, 'Charlie': 95}
```

---

5. Count Word Occurrences Given a list of words:

```
words = ['apple', 'banana', 'apple', 'orange', 'banana', 'apple']
```

Use a dictionary to count how many times each word appears.

---

6. Nested Dictionaries – Student Records Create a dictionary `students` with three students. Each student should have their own dictionary with `'age'` and `'marks'` as keys.

---

7. Retrieve All Keys and Values Given any dictionary, write code to print all the keys and all the values (each on a new line).

---

8. Dictionary from Two Lists Given two lists:

```
keys = ['id', 'name', 'email']
values = [1, 'Sarah', 'sarah@example.com']
```

Create a dictionary that combines them.

Sets

9. Remove Duplicates from a List Given a list with duplicate numbers, convert it into a set to remove duplicates, and then print the result as a list.

---

10. Set Operations – Union and Intersection Given two sets of numbers, find their union and intersection:

```
a = {1, 3, 5, 7}
b = {3, 4, 5, 6}
```

---

11. Check Membership Write code to check if the element `10` is present in the set

```
numbers = {2, 4, 6, 8, 10, 12}
```

---

12. Add and Remove Set Elements Create a set called `animals` with values `'cat'`, `'dog'`, and `'parrot'`. Add `'hamster'` and remove `'dog'`.

---

13. Common Elements in Two Lists Given two lists, use sets to find which elements are common:

```
list1 = [2, 4, 6, 8, 10]
list2 = [3, 6, 9, 12, 15]
```

Combined / Application

14. Unique Words in a Sentence Write a program that takes a sentence from the user, splits it into words, and prints all the unique words (ignore case).

---

15. Frequency Dictionary from a String Given a string, create a dictionary that counts how many times each letter appears (ignore spaces and case).

Example input:

```
sentence = "Python Programming"
```

16. Word Frequency Counter

Write a function that receives a string as input and returns a dictionary where the keys are words and the values are the number of times each word appears in the string. Ignore case and punctuation. Use a loop to build the dictionary.

17. Student Grades Analyzer

Given a dictionary where the keys are student names and the values are lists of their grades (integers), write a function to create a new dictionary where each student name maps to their average grade. Use a loop to calculate averages.

Lesson 08

I. Hands-On Coding- Breaking Down Problems With Functions

VI. Hands-on Coding: Breaking Down Problems with Functions

Goal

- Practice splitting a complex problem into smaller, manageable functions.
- Strengthen skills in writing, calling, testing, and debugging functions.

A. Example Problem 1: User Data Validation and Greeting

Scenario: Write a program that asks the user for their name and age, validates the input, and then greets the user if the data is valid.

Step 1: Break Into Functions

1. **get_user_input** – Get input from the user.
2. **validate_name** – Ensure the name isn't empty and contains only letters.
3. **validate_age** – Ensure age is a positive integer.
4. **greet_user** – Print a greeting message.

Step 2: Code Structure & Examples

```
def get_user_input():
    name = input("Enter your name: ")
    age = input("Enter your age: ")
    return name, age

def validate_name(name):
    return name.isalpha() and len(name) > 0

def validate_age(age):
    return age.isdigit() and int(age) > 0

def greet_user(name, age):
    print(f"Hello, {name}! You are {age} years old.")

def main():
    name, age = get_user_input()
    if not validate_name(name):
        print("Invalid name. Please use only letters.")
        return
    if not validate_age(age):
        print("Invalid age. Please enter a positive number.")
        return
    greet_user(name, age)

main()
```

Explanation:

- Each function does one job—making the code clear and reusable.
- `main()` acts as the program's "controller," coordinating the logic.

B. Example Problem 2: Simple Text Processing

Scenario: Given a sentence, count the number of words and the number of vowels.

Step 1: Break Into Functions

1. **get_sentence** – Get the sentence from the user.
2. **count_words** – Count words in the sentence.
3. **count_vowels** – Count vowels in the sentence.
4. **display_results** – Print the results.

Step 2: Code Structure & Examples

```
def get_sentence():
    return input("Enter a sentence: ")

def count_words(sentence):
    words = sentence.split()
    return len(words)

def count_vowels(sentence):
    vowels = 'aeiouAEIOU'
    return sum(1 for char in sentence if char in vowels)

def display_results(word_count, vowel_count):
    print(f"Words: {word_count}")
    print(f"Vowels: {vowel_count}")

def main():
    sentence = get_sentence()
    word_count = count_words(sentence)
    vowel_count = count_vowels(sentence)
    display_results(word_count, vowel_count)

main()
```

Explanation:

- The problem is split into logical parts, each handled by a function.
- This structure is easier to test, debug, and modify.

C. Tips for Students

- **Plan before you code:** Write down the steps of your solution, then turn each step into a function.
- **Test as you go:** Test each function separately before integrating them.
- **Debugging:** If your program doesn't work, test each function individually to find where the error is.
- **Reuse:** Functions can be reused in other programs or other parts of your code.

D. Group or Individual Work Structure

- **Step 1:** Choose a problem (instructor may assign or let students pick).
- **Step 2:** Break the problem into functions—write down or discuss function names and their jobs.
- **Step 3:** Write each function, testing as you go.
- **Step 4:** Integrate all functions into a `main()` function and run the whole program.
- **Step 5:** Debug any issues together.

E. Sample Problems to Assign

- A calculator that can add, subtract, multiply, and divide (as in the homework!).
- Email validator: Ask for an email, validate its format, and print a result.
- Simple password checker: Get password, check length and characters, print result.
- Word counter: Input a paragraph, count sentences, words, and characters.

Key Understanding

- **Divide and conquer:** Complex problems become much easier when split into smaller pieces.
- **Functions are building blocks:** Each function should have one clear job.
- **Testing and debugging:** With small functions, it's easier to find and fix mistakes.

Ii. Defining Functions In Python

II. Defining Functions in Python

1. Why Do We Need Functions?

- **Explanation:** Functions help break code into reusable, manageable pieces. They make programs easier to write, read, debug, and maintain.
- **Key Understanding:** Functions are like mini-programs inside your code; you give them some data (arguments), they do something, and (optionally) give something back (return value).

---

2. **Syntax: `def` Keyword and Indentation**

- **Explanation:** In Python, functions are defined using the `def` keyword, followed by the function name and parentheses.
- **Python Standard:** PEP 8 – Python's official style guide.
- **Indentation:** Code inside the function must be indented (usually 4 spaces).
- **Example:**

```
def greet():
    print("Hello, world!")
```

- **Key Understanding:** Indentation is critical in Python. Without proper indentation, code will not run.

---

3. **Arguments: Positional, Default, and Keyword Arguments**

- **Positional Arguments:** Passed by position (order matters).
- **Default Arguments:** Provide default values; if not provided by caller, the default is used.
- **Keyword Arguments:** Passed by name, so order doesn't matter.
- **Why Needed:** They make functions flexible and easier to use in different scenarios.
- **Example:**

```
def add(a, b=0):  # 'b' has a default value
    return a + b

# Calling with positional arguments:
print(add(5, 3))     # Output: 8

# Calling with one positional argument, one default:
print(add(5))        # Output: 5

# Using keyword argument:
print(add(a=10, b=2))  # Output: 12
```

- **Key Understanding:** Understand the difference and know when to use each for clarity and flexibility.

---

4. **Naming Conventions: Snake_case, Descriptive Names**

- **Python Standard:** Follow PEP 8.
- **Explanation:** Use lowercase with underscores for function names. Names should describe what the function does.
- **Examples:**

  - Good: `calculate_area()`, `send_email()`
  - Bad: `CalcArea()`, `sendEmail`, `foo()`

- **Why Needed:** Good naming makes code readable and self-explanatory.
- **Key Understanding:** Choose names that reveal intent; avoid abbreviations unless common.

---

5. Docstrings: Using `__doc__` for Documentation

- **Explanation:** The first string inside a function is a *docstring*—it describes what the function does.
- **Python Standard:** Use triple quotes (`"""`) for multi-line docstrings.
- **Why Needed:** Helps others (and your future self) understand your code; used by Python's help system.
- **Example:**

```python
def multiply(a, b):
    """
    Multiply two numbers and return the result.

    Args:
        a (int or float): First number.
        b (int or float): Second number.

    Returns:
        int or float: The result of multiplication.
    """
    return a * b

print(multiply.__doc__)
```

- **Key Understanding:** Always document what your function does, what arguments it expects, and what it returns.

---

6. Calling Functions: How to Invoke a Function

- **Explanation:** To use (call) a function, write its name followed by parentheses and pass required arguments.
- **Example:**

```python
def say_hello(name):
    """Greet the person by name."""
    print(f"Hello, {name}!")

say_hello("Alice")   # Output: Hello, Alice!
```

- **Why Needed:** This is how you execute the code inside your function.
- **Key Understanding:** Function must be defined before it is called. Arguments passed during the call are matched to parameters in the definition.

Summary Table:

Concept	Python Standard / Style	Why It Matters	Example
`def` & Indent	PEP 8	Structure, readability, avoids errors	`def my_func():`
Arguments	PEP 8	Flexibility, function reusability	`def f(a, b=2):`
Naming	PEP 8 (snake_case)	Clarity, teamwork, self-explanatory code	`calculate_sum()`
Docstrings	PEP 257	Documentation, helps others understand code	`"""Add two numbers."""`
Calling Functions	—	Actually uses the function in your code	`my_func()`

Key Understanding for the start

- Functions let you reuse and organize code.
- Proper naming and documentation make your code much easier to read and maintain.
- Following standards (PEP 8, PEP 257) is good practice for professional code.
- Always test your functions after defining them.

Iii. Function Arguments, Return Values, And Scope

III. Function Arguments, Return Values, and Scope

---

1. Arguments: Required vs Optional Arguments

Required (Positional) Arguments

- **Explanation:** Must be provided when the function is called.
- **Example:**

```
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")  # Correct
# greet()      # Error: missing required argument
```

- **Good Practice:** Keep functions simple; use required arguments for essential data.

Optional (Default) Arguments

- **Explanation:** Have default values. If not provided, Python uses the default.
- **Example:**

```
def greet(name, greeting="Hello"):
    print(f"{greeting}, {name}!")

greet("Bob")                 # Output: Hello, Bob!
greet("Bob", "Good morning") # Output: Good morning, Bob!
```

- **Good Practice:** Place required arguments first, then optional ones.
- **Why Needed:** Makes functions more flexible and easier to use in more situations.

Keyword Arguments

- **Explanation:** Arguments passed by name. Improves clarity.
- **Example:**

```
greet(name="Charlie", greeting="Hi")  # Output: Hi, Charlie!
```

---

2. Return Values: Using `return`, Multiple Return Values

Using `return`

- **Explanation:** The `return` statement sends a value back to the caller.
- **Example:**

```
def add(a, b):
    return a + b

result = add(2, 3)
print(result)  # Output: 5
```

- **Good Practice:** Always use `return` when your function needs to send data back.

Multiple Return Values

- **Explanation:** Python can return multiple values as a tuple.
- **Example:**

```
def get_stats(numbers):
    total = sum(numbers)
    count = len(numbers)
    average = total / count if count > 0 else 0
    return total, count, average

t, c, avg = get_stats([1, 2, 3])
```

```
print(f"Total: {t}, Count: {c}, Average: {avg}")
# Output: Total: 6, Count: 3, Average: 2.0
```

- **Good Practice:** Use multiple returns when a function naturally produces several results.

No Return (Implicit `None`)

- **Explanation:** If a function doesn't return anything, Python returns `None`.
- **Example:**

```
def say_hello():
    print("Hello!")

result = say_hello()
print(result)  # Output: None
```

---

3. Scope: Local vs Global Variables, `global` Keyword, Best Practices

Local Variables

- **Explanation:** Variables defined inside a function are *local* to that function.
- **Example:**

```
def foo():
    x = 5  # x is local to foo()
    print(x)

foo()
# print(x)  # Error: x is not defined outside foo()
```

Global Variables

- **Explanation:** Variables defined outside any function are *global* and can be accessed inside functions (but not modified unless declared global).
- **Example:**

```
y = 10  # global variable

def bar():
    print(y)  # Can access global y

bar()
```

Modifying Globals with `global` Keyword

- **Explanation:** To modify a global variable inside a function, use `global`.
- **Example:**

```python
count = 0

def increment():
    global count
    count += 1

increment()
print(count)  # Output: 1
```

- **Good Practice: Avoid using global variables unless absolutely necessary**—they make code harder to understand and debug. Prefer returning values and passing them as arguments.

Best Practices for Scope

- Prefer local variables—limit their "reach."
- Use function arguments and return values to share data between functions.
- Avoid using `global` unless there's a clear, justifiable reason.

Summary Table

Concept	What Is It?	Example/Practice	Good Practice
Required Arg	Must be provided	`def f(x): ...`	Use for essential info
Optional Arg	Has default value	`def f(x, y=0): ...`	Put after required args
Return Value	Data sent back to caller	`return x + y`	Always use if output needed
Multiple Return	Return several values as tuple	`return x, y`	Use for related data
Local Scope	Exists only inside the function	`def f(): x = 1`	Use for temporary values
Global Scope	Exists everywhere, use sparingly	`global x`	Avoid modifying globally

---

Key Understandings for Beginners

- Understand which arguments are required and which are optional.
- Use `return` to get data out of your function.
- Know the difference between local and global variables.
- Favor local variables and return values for safer, more modular code.

Iiv. Homework

VIII. Homework Assignment

Task

Create a Python program with multiple functions that simulate a simple calculator.

Requirements:

- Use separate functions for addition, subtraction, multiplication, and division.
- Include user input handling.
- Implement error handling (such as dividing by zero or invalid input).
- Use a main function to organize the workflow.
- Display results to the user.

Example Solution

Program Structure

1. **add(a, b)** – returns sum
2. **subtract(a, b)** – returns difference
3. **multiply(a, b)** – returns product
4. **divide(a, b)** – returns quotient (handle division by zero)
5. **get_numbers()** – gets two numbers from the user, handles invalid input
6. **main()** – controls the calculator loop and user interaction

Example Code

```
def add(a, b):
    """Return the sum of a and b."""
    return a + b

def subtract(a, b):
    """Return the difference of a and b."""
    return a - b

def multiply(a, b):
    """Return the product of a and b."""
    return a * b

def divide(a, b):
    """Return the division of a by b. Handles division by zero."""
    try:
        return a / b
    except ZeroDivisionError:
        print("Error: Cannot divide by zero!")
        return None

def get_numbers():
```

```
    """Get two numbers from the user, handling invalid input."""
    while True:
        try:
            a = float(input("Enter the first number: "))
            b = float(input("Enter the second number: "))
            return a, b
        except ValueError:
            print("Invalid input. Please enter valid numbers.")

def main():
    print("Simple Calculator")
    print("Select operation: +, -, *, /")
    operations = {'+': add, '-': subtract, '*': multiply, '/': divide}

    while True:
        op = input("Enter operation (+, -, *, /) or 'q' to quit: ")
        if op == 'q':
            print("Goodbye!")
            break
        if op not in operations:
            print("Invalid operation. Try again.")
            continue

        a, b = get_numbers()
        result = operations[op](a, b)
        if result is not None:
            print(f"Result: {result}")

if __name__ == "__main__":
    main()
```

Explanation

- **Separation of Concerns:** Each operation is handled by its own function, making code organized and reusable.
- **Error Handling:**
  - `get_numbers()` checks for valid number input.
  - `divide()` handles division by zero.
- **User Interaction:** The user can repeatedly perform calculations or quit.
- **Scalability:** More operations or features (like exponentiation) can be added easily by defining new functions.

Good Practices Illustrated

- Each function has a single responsibility.
- Functions use docstrings for documentation.
- Errors are handled gracefully.
- User input is validated.
- Code is modular and easy to read.

Iv. Advanced Functions- Decorators, Anonymous, And Lambda Functions

IV. Advanced Functions: Decorators, Anonymous, and Lambda Functions

---

1. Decorators

What Are Decorators?

- **Explanation:** Decorators are special functions in Python that "wrap" or modify the behavior of other functions without changing their actual code.
- **Why Use Them?** They help you add common functionality—like logging, access control, or timing—easily and cleanly.
- **Beginner Analogy:** Think of a decorator like a phone case: the phone works the same, but the case adds extra features (like protection or color) without changing the phone itself.

Basic Syntax

- A decorator is applied to a function using the `@decorator_name` syntax, just above the function definition.
- **Example:**

```
def simple_decorator(func):
    def wrapper():
        print("Before function runs")
        func()
        print("After function runs")
    return wrapper

@simple_decorator
def say_hello():
    print("Hello!")

say_hello()
# Output:
# Before function runs
# Hello!
# After function runs
```

Common Use-Cases

- **Logging:** Recording when a function runs.

- **Timing:** Measuring how long a function takes to run.
- **Access Control:** Checking permissions before running a function.

Good Practices

- Use decorators to avoid repeating code (DRY principle).
- Always use `functools.wraps` if you want to preserve the decorated function's name and docstring (more advanced, but good to mention).
- Don't overuse decorators—keep code readable.

---

2. Anonymous/Lambda Functions

What Are Lambda Functions?

- **Explanation:** A lambda function is a small, anonymous function (i.e., it has no name). Used for short, simple operations, often passed as arguments to other functions.
- **Syntax:**

```
lambda arguments: expression
```

Where to Use Them

- When you need a short function for a short period—usually as an argument to functions like `map()`, `filter()`, or `sorted()`.

Examples A. Simple Lambda Example

```
add = lambda x, y: x + y
print(add(2, 3))  # Output: 5
```

B. Using Lambda with `sorted()`

```
names = ['Anna', 'John', 'Mike']
# Sort by the last letter
sorted_names = sorted(names, key=lambda name: name[-1])
print(sorted_names)  # Output: ['Anna', 'Mike', 'John']
```

C. Using Lambda with `filter()`

```
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # Output: [2, 4, 6]
```

Good Practices

- Use lambda functions for simple, one-line tasks.
- If the function gets complex, use a `def` instead for readability.
- Name lambda functions only if you must reuse them; otherwise, use them inline.

Summary Table

Concept	What Is It?	Example	Good Practice
Decorator	Wraps/modifies another function	`@my_decorator`	Use for code reuse, keep them readable
Lambda	Small, anonymous, one-line function	`lambda x: x*2`	Use for simple, short-lived operations

---

Key Takeaways for Beginners

- **Decorators:** Allow you to add extra features to your functions with clean syntax.
- **Lambda Functions:** Handy for quick, throwaway functions, especially as arguments.
- **Best Practice:** Use both features to write more modular, concise, and readable code—but don't sacrifice clarity for cleverness!

Outline

Lesson 8: Python Functions

Duration: 2 hours

I. Introduction & Recap (10 min)

- **Recap:** Review of previous lesson's data structures (lists, tuples, sets, dicts)
- **Discussion:** Why functions are important in programming

II. Defining Functions in Python (20 min)

- **Syntax:** `def` keyword, indentation
- **Arguments:** Positional, default, keyword arguments
- **Naming Conventions:** Snake_case, descriptive names
- **Docstrings:** Using `__doc__` for documentation
- **Calling Functions:** How to invoke a function

III. Function Arguments, Return Values, and Scope (20 min)

- **Arguments:** Required vs optional arguments
- **Return Values:** Using `return`, multiple return values
- **Scope:** Local vs global variables, `global` keyword, best practices

---

IV. Advanced Functions: Decorators, Anonymous, and Lambda Functions (20 min)

- **Decorators:** What are they, basic syntax, common use-cases
- **Anonymous/Lambda Functions:** Syntax, where to use them

V. Introduction to Regular Expressions (20 min)

- **What are regular expressions?**
- **Basic syntax in Python:** `import re`, pattern matching
- **Examples:** Finding patterns in strings

VI. Hands-on Coding: Breaking Down Problems with Functions (20 min)

- **Exercise:** Break a complex problem (e.g., data validation, text processing) into multiple functions
- **Group or Individual Work:** Write, test, and debug functions

---

VII. Q&A and Wrap-Up (10 min)

- Clarify doubts
- Highlight importance of modular code

VIII. Homework Assignment

- **Task:** Create a Python program with multiple functions that simulate a simple calculator (addition, subtraction, multiplication, division, error handling, etc.).

Lesson 09

Exercise

LESSON 09: Additional Python Function exercises (with solutions)

---

16. Temperature Converter Write a function `celsius_to_fahrenheit(c)` that converts Celsius to Fahrenheit using the formula: `F = C * 9/5 + 32`.

---

17. BMI Calculator Create a function `calculate_bmi(weight, height)` that returns the BMI value and a health message.

---

18. Password Validator Write a function `is_valid_password(password)` that checks if the password is at least 8 characters long and includes both letters and numbers.

---

19. Tip Calculator Define a function `calculate_tip(bill_amount, tip_percent=15)` to return total tip and final amount.

---

20. Email Masker Write a function `mask_email(email)` that masks an email (e.g., j***e@gmail.com).

---

21. Age Calculator Write a function `calculate_age(birth_year, current_year)` that returns the person's age.

---

22. To-Do List Manager (Add Task) Create a function `add_task(todo_list, task)` that adds a task to a list.

---

23. Loan Monthly Payment Calculator Write a function `calculate_monthly_payment(principal, rate, years)` using the formula for fixed monthly payments.

Solutions

16. Temperature Converter

```
def celsius_to_fahrenheit(c):
    return c * 9/5 + 32

print(celsius_to_fahrenheit(25))
```

17. BMI Calculator

```
def calculate_bmi(weight, height):
    bmi = weight / (height ** 2)
    if bmi < 18.5:
        return bmi, "Underweight"
    elif bmi < 25:
        return bmi, "Normal"
    elif bmi < 30:
        return bmi, "Overweight"
    else:
        return bmi, "Obese"
```

```
bmi_value, status = calculate_bmi(70, 1.75)
print(f"BMI: {bmi_value:.2f}, Status: {status}")
```

18. Password Validator

```
def is_valid_password(password):
    has_letter = any(c.isalpha() for c in password)
    has_digit = any(c.isdigit() for c in password)
    return len(password) >= 8 and has_letter and has_digit

print(is_valid_password("abc12345"))
```

---

19. Tip Calculator

```
def calculate_tip(bill_amount, tip_percent=15):
    tip = bill_amount * (tip_percent / 100)
    return tip, bill_amount + tip

tip, total = calculate_tip(100)
print(f"Tip: {tip}, Total: {total}")
```

20. Email Masker

```
def mask_email(email):
    name, domain = email.split('@')
    masked = name[0] + '***' + name[-1]
    return masked + '@' + domain

print(mask_email("johndoe@gmail.com"))
```

21. Age Calculator

```
def calculate_age(birth_year, current_year):
    return current_year - birth_year

print(calculate_age(1990, 2025))
```

---

22. To-Do List Manager (Add Task)

```
def add_task(todo_list, task):
    todo_list.append(task)
    return todo_list

my_list = []
my_list = add_task(my_list, "Finish homework")
print(my_list)
```

23. Loan Monthly Payment Calculator

```
def calculate_monthly_payment(principal, rate, years):
    monthly_rate = rate / 12 / 100
    months = years * 12
    payment = principal * monthly_rate / (1 - (1 + monthly_rate) ** -months)
    return payment

print(round(calculate_monthly_payment(10000, 5, 2), 2))
```

Exercise

Lesson 09: Python Functions (Exercise)

Each assignment is designed to reinforce the core concepts covered in **Lesson 8: Python Functions**, which typically include:

- Function definition and calling
- Parameters and arguments
- Return statements
- Scope (local/global)
- Default and keyword arguments
- Nested functions
- Recursion (if applicable)

---

1. Basic Function Definition Write a function called `greet_user()` that prints "Hello, welcome to Python functions!"

Goal: Ensure the learner can define and call a simple function with no parameters.

---

2. Function with One Parameter Define a function `greet(name)` that prints a personalized greeting like "Hello, Alice!"

Goal: Practice using parameters and string formatting.

---

3. Function with Return Statement Create a function `square(num)` that returns the square of the number provided.

Goal: Understand how return statements work and how to use returned values.

---

4. Multiple Parameters Write a function `add_numbers(a, b)` that returns the sum of two numbers.

Goal: Use multiple parameters and arithmetic operations.

---

5. Default Arguments Define a function `power(base, exponent=2)` that returns base raised to the exponent. Test it with and without the second argument.

Goal: Practice using default values in function parameters.

---

6. Keyword Arguments Call a function `describe_pet(animal_type, pet_name)` using keyword arguments in different orders.

Goal: Reinforce understanding of how keyword arguments work.

---

7. Positional vs Keyword Arguments Modify `describe_pet()` to demonstrate the difference between positional and keyword arguments.

---

8. Local vs Global Scope Write a function `show_scope()` that tries to modify a global variable. Then use `global` keyword to actually modify it.

Goal: Illustrate the difference between local and global variables.

---

9. Function that Returns Multiple Values Write a function `split_name(full_name)` that returns first and last name separately.

---

10. Nested Functions Define a function `outer()` with an inner function `inner()` inside it. The outer function should call the inner one.

Goal: Understand function nesting and scope implications.

---

11. Calculator with Functions Create a simple calculator with functions: `add`, `subtract`, `multiply`, and `divide`. Each should take two numbers.

Goal: Combine multiple concepts (functions, parameters, return) into one task.

---

12. Recursive Function Write a recursive function `factorial(n)` that returns the factorial of `n`.

Goal: Introduce the concept of recursion and base cases.

---

13. Function with a List Argument Write a function `print_shopping_list(items)` that takes a list and prints each item.

---

14. Function Reusability Write a function `is_even(number)` that returns True if a number is even, else False. Then use it inside another function `filter_evens(numbers)` that returns only even numbers from a list.

---

15. Simple Menu using Functions Create a text menu with three options (e.g., greet, add two numbers, quit). Implement each option using a function and use a loop to keep the program running.

Goal: Integrate functions into a user-interactive program and demonstrate modular design.

Arbitrary number of arguments varying number of arguments can be passed to a function using *args and **kwargs

*args = allows passing a variable number of positional arguments (non-keyword arguments)

**kwargs = allows passing a variable number of keyword arguments (key-value pairs)

- unpacking operator (can be used to unpack a list or tuple into positional arguments)

```
def sum_123(a,b,):
    return a +b

print(sum_123(1,23))
print(sum_123(1,2,3,4)) # error: sum_123() takes 2 positional arguments but 4 were given

def sum_all(*args) -> int:
    """Return the sum of all arguments."""
    return sum(args)
```

convention is to use *args for positional arguments but you can use any name, but* args is a common convention

```
def sum_all2(*nums) -> int:
    """Return the sum of all arguments."""
    return sum(nums)

print(sum_all(1, 2, 3))
print(sum_all(1, 2, 3, 4, 5,43,43,34,3,423,423,423423))


print(sum_all2(1, 2, 3))
print(sum_all2(1, 2, 3, 4, 5,43,43,34,3,423,423,423))
```

**kwargs allows passing a variable number of keyword arguments (key-value pairs)

```
def print_address(**kwargs) -> None:
    """Print the address."""
```

```
    for key, value in kwargs.items():
        print(f"{key}: {value}")
    print(type(kwargs))  # <class 'dict'>, kwargs is a dictionary

print_address(street="Dol 42", city="LJubljana", zip="1000", country="Slovenia")
```

order of arguments in function definition is crucial:

- 1. positional arguments
- 2. *args (variable number of positional arguments)*
- 3. keyword arguments (default arguments)

```
def shipping_label(*args, **kwargs):
    pass

print(shipping_label("Alex", "Castro",
                     street="Dol 42",
                     city="LJubljana",
                     zip="1000",
                     country="Slovenia"))
```

Lesson 09- Python Functions (Solutions)

Lesson 09: Python Functions (Solutions)

1. Basic Function Definition

```
def greet_user():
    print("Hello, welcome to Python functions!")

greet_user()
```

---

2. Function with One Parameter

```
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
```

---

3. Function with Return Statement

```
def square(num):
    return num * num

result = square(5)
print(result)  # Output: 25
```

---

4. Multiple Parameters

```
def add_numbers(a, b):
    return a + b

print(add_numbers(3, 7))  # Output: 10
```

---

5. Default Arguments

```
def power(base, exponent=2):
    return base ** exponent

print(power(3))      # Output: 9
print(power(3, 3))   # Output: 27
```

---

6. Keyword Arguments

```
def describe_pet(animal_type, pet_name):
    print(f"I have a {animal_type} named {pet_name}.")

describe_pet(animal_type="dog", pet_name="Rex")
describe_pet(pet_name="Whiskers", animal_type="cat")
```

---

7. Positional vs Keyword Arguments

```
def describe_pet(animal_type, pet_name):
    print(f"My {animal_type}'s name is {pet_name}.")

# Positional
describe_pet("hamster", "Nibbles")

# Keyword
describe_pet(pet_name="Bubbles", animal_type="fish")
```

---

8. Local vs Global Scope

```
x = 5

def show_scope():
    x = 10  # Local variable
    print("Inside function (local x):", x)

show_scope()
print("Outside function (global x):", x)
```

```
def modify_global():
    global x
    x = 20

modify_global()
print("After global modification:", x)
```

---

9. Function that Returns Multiple Values

```
def split_name(full_name):
    parts = full_name.split()
    return parts[0], parts[1]

first, last = split_name("Ada Lovelace")
print("First:", first)
print("Last:", last)
```

---

10. Nested Functions

```
def outer():
    print("Inside outer function.")

    def inner():
        print("Inside inner function.")

    inner()

outer()
```

---

11. Calculator with Functions

```
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b

def divide(a, b):
    if b == 0:
        return "Error: Division by zero!"
```

```
    return a / b

print(add(4, 2))
print(subtract(4, 2))
print(multiply(4, 2))
print(divide(4, 2))
```

---

12. Recursive Function

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(5))  # Output: 120
```

---

13. Function with a List Argument

```
def print_shopping_list(items):
    for item in items:
        print(f"- {item}")

shopping = ["Milk", "Bread", "Eggs"]
print_shopping_list(shopping)
```

---

14. Function Reusability

```
def is_even(number):
    return number % 2 == 0

def filter_evens(numbers):
    evens = []
    for num in numbers:
        if is_even(num):
            evens.append(num)
    return evens

print(filter_evens([1, 2, 3, 4, 5, 6]))  # Output: [2, 4, 6]
```

---

15. Simple Menu using Functions

```
def greet():
    print("Hello from the menu!")

def add_numbers():
    a = int(input("Enter first number: "))
    b = int(input("Enter second number: "))
    print("Sum:", a + b)

def main_menu():
    while True:
        print("\nMenu:")
        print("1. Greet")
        print("2. Add two numbers")
        print("3. Quit")
        choice = input("Choose an option: ")

        if choice == '1':
            greet()
        elif choice == '2':
            add_numbers()
        elif choice == '3':
            print("Goodbye!")
            break
        else:
            print("Invalid option. Try again.")

main_menu()
```

Lesson 10

I. Introduction To Pandas

I. Introduction to Pandas

What Are Packages in Python? In Python, **packages** are collections of modules that provide additional functionality. Instead of writing everything from scratch, we can use packages that others have written and shared.

- Think of packages as **toolboxes** — they contain useful tools (functions, classes, methods) for specific types of tasks.
- For data manipulation and analysis, two of the most commonly used packages are **Pandas** and **NumPy**.

Installing Packages To use a package, you may need to install it first using `pip` (Python's package installer):

```
pip install pandas
pip install numpy
```

In Jupyter notebooks or IPython, use:

```
!pip install pandas
!pip install numpy
```

Importing Packages Once installed, import them at the beginning of your script or notebook:

```
import pandas as pd
import numpy as np
```

`pd` and `np` are **aliases** commonly used by the Python community for convenience.

What is Pandas and Why It's Useful? **Pandas** is a powerful, open-source Python package used for data manipulation and analysis. It provides flexible data structures that make it easy to clean, transform, explore, and visualize datasets — particularly **tabular data** (like spreadsheets or SQL tables).

Why Use Pandas?

- Allows you to **load data** from CSV, Excel, SQL, etc.
- Helps perform **filtering, sorting, grouping, and aggregating** operations easily
- Designed for **fast and intuitive** data exploration
- Seamlessly integrates with **NumPy, Matplotlib**, and other libraries

---

Key Data Structures in Pandas Pandas is built on two main data structures:

A-> Series: One-dimensional labeled array

- Like a single column of data with labels (index)
- Stores any data type: numbers, strings, booleans, etc.

Example:

```
import pandas as pd

data = [10, 20, 30, 40]
labels = ['a', 'b', 'c', 'd']

s = pd.Series(data, index=labels)
print(s)
```

Output:

```
a    10
b    20
c    30
d    40
dtype: int64
```

B-> DataFrame: Two-dimensional labeled data structure

- Think of it as an **Excel table** or a **SQL table**: rows and columns, with labels
- Each column is a Series
- Most common structure used in Pandas

Example:

```
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['NYC', 'LA', 'Chicago']
}

df = pd.DataFrame(data)
print(df)
```

Output:

```
      Name  Age     City
0    Alice   25      NYC
1      Bob   30       LA
2  Charlie   35  Chicago
```

How is Pandas Different from NumPy? While **NumPy** is great for numerical arrays and mathematical operations, it is not designed for **labeled** or **heterogeneous** (mixed-type) data.

Feature	NumPy (ndarray)	Pandas (DataFrame/Series)
Best For	Numerical data	Labeled, structured/tabular data
Supports Labels		Row and column labels
Mixed Data Types	(same type only)	Different types in each column
Readability	Moderate	High (like spreadsheets)
Data Source Loading	Limited	Easy load from CSV, Excel, SQL, etc.

Use Pandas when you want to:

- Handle real-world datasets
- Work with structured (table-like) data
- Do analysis or cleaning before feeding data into a model

I.II. Introduction to Polars (an alternative to Pandas) **

Introduction to **Polars** – A Fast Modern Alternative to Pandas

What is Polars? **Polars** is a newer Python library designed for fast, efficient, and parallelized data manipulation. It's written in **Rust**, which makes it **significantly faster** than Pandas in many cases, especially when handling large datasets.

Why Use Polars?

- **Performance:** Built for speed with multi-threaded execution
- **Lazy Evaluation:** Like SQL or Spark, operations are planned and optimized before execution
- **Memory-efficient:** Uses Arrow-based columnar storage
- **Familiar Syntax:** Inspired by Pandas, but with its own approach

---

Installing and Importing Polars

```
pip install polars
```

```
import polars as pl
```

Polars vs. Pandas

Feature	Pandas	Polars
Language Core	Python	Rust
Speed	Moderate	Very Fast (multithreaded)
Evaluation	Eager (immediate execution)	Lazy (deferred until needed)
API Design	Broad and flexible	Minimal and efficient
Memory Usage	Higher	Lower
Syntax Style	Pythonic (object-based)	Functional (method chaining)

Example: Creating a DataFrame in Polars

```
import polars as pl

df = pl.DataFrame({
    "Name": ["Alice", "Bob", "Charlie"],
    "Age": [25, 30, 35],
    "City": ["NYC", "LA", "Chicago"]
})

print(df)
```

Output:

```
shape: (3, 3)

 Name      Age   City
```

```
---       ---   ---
str       i64   str

Alice      25   NYC
Bob        30   LA
Charlie    35   Chicago
```

Selecting and Filtering in Polars

```python
# Select a single column
df.select("Name")

# Filter rows
df.filter(pl.col("Age") > 28)
```

When to Use Polars

- Working with **very large datasets** where performance is a bottleneck
- When you need **parallel processing** or **lazy evaluation**
- For production environments where **speed and memory efficiency** matter

  Note: Polars is newer, so while it's powerful, its ecosystem isn't as mature as Pandas yet. It's ideal for power users or developers who are performance-focused.

Pandas vs. Polars: Side-by-Side Cheat Sheet

Task	Pandas	Polars
Import package	`import pandas as pd`	`import polars as pl`
Create DataFrame	`pd.DataFrame({...})`	`pl.DataFrame({...})`
Read CSV	`pd.read_csv('file.csv')`	`pl.read_csv('file.csv')`
View top rows	`df.head()`	`df.head()`
Select column	`df['col']` or `df.col`	`df.select("col")`
Select multiple columns	`df[['col1', 'col2']]`	`df.select(["col1", "col2"])`
Filter rows	`df[df['Age'] > 30]`	`df.filter(pl.col("Age") > 30)`
Add new column	`df['new'] = df['Age'] + 5`	`df.with_columns((pl.col("Age") + 5).alias("new"))`
Drop column	`df.drop(columns=['col'])`	`df.drop("col")`
Group by + aggregation	`df.groupby('City').mean()`	`df.groupby("City").agg(pl.col("Age").mean())`
Describe data	`df.describe()`	`df.describe()` *(limited in Polars)*
Sort values	`df.sort_values('Age')`	`df.sort("Age")`
Shape of DataFrame	`df.shape`	`df.shape`
Convert to NumPy	`df.to_numpy()`	`df.to_numpy()` *(as NumPy-compatible)*

Task	Pandas	Polars
Lazy execution support	*(Not supported)*	`pl.scan_csv()` for lazy pipelines
Multithreading	*(Single-threaded by default)*	*(Multithreaded by design)*
Performance with large data	Slower	Much faster

Summary

Aspect	Pandas	Polars
Language Base	Python	Rust (Python API)
Evaluation Model	Eager	Lazy (optional)
Speed/Performance	Good	Excellent for large data
API Style	Pythonic, imperative	Functional, method-chaining
Learning Curve	Easier for beginners	Slightly steeper
Ecosystem Maturity	Very mature	Rapidly growing
Best Use Case	General-purpose data analysis	Large-scale, high-performance tasks

Ii. Loading Data Into Pandas

II. Loading Data into Pandas

Data analysis often starts with **loading external datasets**—typically stored in files like CSV or Excel. Pandas makes this process very easy and intuitive with built-in functions.

Why Load External Data?

Real-world data doesn't usually come from variables typed into code. Instead, data is collected and stored in:

- **CSV files** (comma-separated values)
- **Excel spreadsheets**
- **Databases**
- **Web APIs**

To analyze such data, we need to **import** it into our Python program. That's where Pandas shines.

---

1. Reading CSV Files with `pd.read_csv()`

CSV (Comma-Separated Values) files are one of the most common formats for tabular data. Each line in a CSV file represents a row, with values separated by commas.

Syntax:

```
import pandas as pd

df = pd.read_csv("filename.csv")
```

Parameters You Might Use:

- `sep=','`: delimiter (use `'\t'` for TSV files)
- `header=0`: row to use as column names
- `index_col=0`: column to use as the row index
- `na_values=["NA", "?"]`: treat certain strings as missing values

Example:

```
df = pd.read_csv("students_scores.csv")
print(df.head())
```

Sample CSV: `students_scores.csv`

```
Name,Math,Science,English
Alice,85,90,78
Bob,72,80,69
Charlie,88,85,95
```

---

2. Reading Excel Files with `pd.read_excel()`

Excel is widely used in businesses, and Pandas allows reading Excel `.xls` or `.xlsx` files.

Syntax:

```
df = pd.read_excel("filename.xlsx", sheet_name="Sheet1")
```

Example:

```
df = pd.read_excel("sales_data.xlsx", sheet_name="Quarter1")
print(df.head())
```

Sample Excel: `sales_data.xlsx` (Sheet name: `Quarter1`)

Product	Region	Sales
A	East	1000
B	West	850
C	North	920

You can create this manually in Excel or programmatically using:

```python
import pandas as pd

data = {
    "Product": ["A", "B", "C"],
    "Region": ["East", "West", "North"],
    "Sales": [1000, 850, 920]
}
df = pd.DataFrame(data)
df.to_excel("sales_data.xlsx", sheet_name="Quarter1", index=False)
```

---

3. Understanding File Paths and Working Directories

What is a File Path?

- A **file path** is the address where your data file is stored.
- **Relative path**: Refers to a file in the same or nearby folder (e.g., `"data/students.csv"`)
- **Absolute path**: Full address from the system root (e.g., `"C:/Users/Name/Desktop/data.csv"`)

Checking Your Current Working Directory Use this to find or set where Python is looking for files:

```python
import os

print(os.getcwd())  # Current working directory
```

You can also change it:

```python
os.chdir("C:/Users/Name/Documents/MyProject/")
```

Or, use a **relative path** to load files from a subfolder:

```python
df = pd.read_csv("data/students_scores.csv")
```

Tips for Loading Data

- Always check the first few rows using `df.head()`
- Use `df.info()` to check structure and types
- If data isn't displaying correctly, try tweaking parameters like `sep`, `encoding`, or `header`

---

4. Reading JSON Files with `pd.read_json()`

What is a JSON File? JSON (JavaScript Object Notation) is a popular format for storing structured data, especially for APIs and web data. It's a text format that is easy to read and write for both humans and machines.

- JSON structures resemble Python dictionaries and lists.
- It's often used when transferring data between servers and clients or saving complex structured data.

Reading JSON Files in Pandas Pandas can **automatically parse** JSON files into DataFrames with the function:

```
df = pd.read_json("filename.json")
```

Example:

```
import pandas as pd

df = pd.read_json("employees.json")
print(df)
```

Sample JSON: `employees.json`

```
[
  {"Name": "Alice", "Department": "HR", "Salary": 50000},
  {"Name": "Bob", "Department": "Finance", "Salary": 60000},
  {"Name": "Charlie", "Department": "IT", "Salary": 70000}
]
```

Expected Output:

```
      Name Department  Salary
0    Alice         HR   50000
1      Bob    Finance   60000
2  Charlie         IT   70000
```

Generating Sample JSON File in Python To create the file within your script:

```
import json

data = [
    {"Name": "Alice", "Department": "HR", "Salary": 50000},
    {"Name": "Bob", "Department": "Finance", "Salary": 60000},
    {"Name": "Charlie", "Department": "IT", "Salary": 70000}
]

with open("employees.json", "w") as f:
    json.dump(data, f, indent=4)
```

Notes When Working with JSON in Pandas

- JSON can be **nested**. For complex or deeply nested JSON, you may need to **normalize** it using `pd.json_normalize()`:

```
from pandas import json_normalize
json_data = {
    "employees": [
        {"name": "Alice", "details": {"dept": "HR", "salary": 50000}},
        {"name": "Bob", "details": {"dept": "Finance", "salary": 60000}}
    ]
}

df = json_normalize(json_data['employees'], sep="_")
print(df)
```

- Ensure your JSON is **well-formed** and matches the structure Pandas expects.
- Works great for data from **APIs**, logs, or systems that generate structured data.

Sample Files for This Lesson

Include these two datasets in your class or project folder:

1. students_scores.csv

```
Name,Math,Science,English
Alice,85,90,78
Bob,72,80,69
Charlie,88,85,95
```

2. sales_data.xlsx

- Sheet name: `Quarter1`
- Columns: Product, Region, Sales

To generate them in code:

```
# CSV file
df_csv = pd.DataFrame({
    "Name": ["Alice", "Bob", "Charlie"],
    "Math": [85, 72, 88],
    "Science": [90, 80, 85],
    "English": [78, 69, 95]
})
df_csv.to_csv("students_scores.csv", index=False)

# Excel file
df_excel = pd.DataFrame({
    "Product": ["A", "B", "C"],
    "Region": ["East", "West", "North"],
    "Sales": [1000, 850, 920]
})
df_excel.to_excel("sales_data.xlsx", sheet_name="Quarter1", index=False)
```

3. employee.json

```
[
  {"Name": "Alice", "Department": "HR", "Salary": 50000},
  {"Name": "Bob", "Department": "Finance", "Salary": 60000},
  {"Name": "Charlie", "Department": "IT", "Salary": 70000}
]
```

Summary of File Types and Read Functions

File Type	Extension	Pandas Function	Example
CSV	`.csv`	`pd.read_csv()`	`pd.read_csv("data.csv")`
Excel	`.xlsx`	`pd.read_excel()`	`pd.read_excel("file.xlsx")`
JSON	`.json`	`pd.read_json()`	`pd.read_json("file.json")`

Iii. Dataframes And Basic Operations

III. DataFrames and Basic Operations

Once your data is loaded into a **Pandas DataFrame**, the next step is to explore, understand, and manipulate it. This section walks through the **most common operations** you'll use in real-world data analysis.

---

1. Exploring a DataFrame

Understanding the structure of your data is the first step in working with it effectively.

`df.head(n)` Displays the **first n rows** of the DataFrame (default is 5).

```
df.head()
df.head(10)  # First 10 rows
```

`df.info()` Gives a **summary of the DataFrame**, including:

- Number of rows and columns
- Column data types
- Memory usage
- Count of non-null values

```
df.info()
```

`df.describe()` Provides **statistical summary** for numeric columns:

- Count, mean, std, min, max, and quartiles

```
df.describe()
```

Example:

```
import pandas as pd

df = pd.read_csv("students_scores.csv")
print(df.head())
print(df.info())
print(df.describe())
```

Sample `students_scores.csv`:

```
Name,Math,Science,English
Alice,85,90,78
Bob,72,80,69
Charlie,88,85,95
David,91,87,89
Eva,76,90,84
```

---

2. Selecting Data

Pandas makes it easy to select specific rows or columns using different indexing methods.

Select Columns

- **Single column**:

```
df['Math']
```

- **Multiple columns**:

```
df[['Math', 'English']]
```

Select Rows

- **By label (index) using `.loc[]`**:

```
df.loc[0]          # First row
df.loc[0:2]        # Rows with labels 0 to 2 inclusive
df.loc[:, 'Math']  # All rows, only 'Math' column
```

- **By position using `.iloc[]`**:

```
df.iloc[0]          # First row
df.iloc[0:3]        # First three rows
df.iloc[:, 1]       # All rows, second column
```

---

3. Sorting Data

Sorting helps to quickly analyze trends or rank values.

Sort by a single column:

```
df.sort_values(by='Math')
```

Sort in descending order:

```
df.sort_values(by='English', ascending=False)
```

---

4. Filtering with Conditions (Boolean Indexing)

Use **logical expressions** to filter rows.

```
df[df['Math'] > 80]
```

Combine multiple conditions:

```
df[(df['Math'] > 80) & (df['Science'] > 85)]
```

---

5. Adding and Removing Columns

Add a new column:

```
df['Average'] = df[['Math', 'Science', 'English']].mean(axis=1)
```

Remove a column:

```
df.drop('English', axis=1, inplace=True)  # inplace modifies the DataFrame
```

Demo: Combined Operations

```
import pandas as pd

# Load dataset
df = pd.read_csv("students_scores.csv")

# Explore data
print(df.head())
print(df.info())
print(df.describe())

# Select data
print(df[['Name', 'Math']])
print(df.iloc[0:2])
```

```python
# Sort data
sorted_df = df.sort_values(by='Science', ascending=False)
print(sorted_df)

# Filter data
high_scorers = df[df['Math'] > 80]
print(high_scorers)

# Add average score column
df['Average'] = df[['Math', 'Science', 'English']].mean(axis=1)

# Drop a column
df = df.drop('English', axis=1)

print(df)
```

Sample Dataset (CSV): `students_scores.csv`

```
Name,Math,Science,English
Alice,85,90,78
Bob,72,80,69
Charlie,88,85,95
David,91,87,89
Eva,76,90,84
```

Summary Table: Common DataFrame Operations

Operation	Syntax
View first rows	`df.head()`
View structure/info	`df.info()`
View statistics	`df.describe()`
Select one column	`df['Column']`
Select multiple columns	`df[['Col1', 'Col2']]`
Select by label	`df.loc[0:2]`
Select by index	`df.iloc[0:2]`
Sort by column	`df.sort_values(by='Col')`
Filter by condition	`df[df['Col'] > value]`
Add column	`df['New'] = ...`
Remove column	`df.drop('Col', axis=1)`

Iv. Hands-On Practice With Pandas

IV. Hands-on Practice with Pandas

Reinforce core concepts through practical data manipulation tasks using **real-world-style datasets**.

Dataset Sources

We'll use two types of datasets:

1. **Built-in dataset** via `seaborn` (Iris or Titanic)
2. **CSV file** (e.g. `employees.csv` or `students_scores.csv` from earlier)

Step-by-Step Practice Tasks

---

1. Loading a Dataset

Example A – Load Titanic dataset from `seaborn`:

```
import seaborn as sns
import pandas as pd

df = sns.load_dataset('titanic')
df.head()
```

Example B – Load a CSV file:

```
df = pd.read_csv("employees.csv")
df.head()
```

Sample `employees.csv`:

```
Name,Department,Salary,Experience,Location
Alice,HR,50000,3,New York
Bob,Finance,60000,5,Chicago
Charlie,IT,70000,8,San Francisco
Daisy,Finance,58000,,Boston
Edward,IT,72000,7,Chicago
```

---

2. Column Selection

Why? You often only need a subset of columns for your analysis or modeling.

```
df[['Name', 'Salary']]
```

Use column selection to narrow your focus to relevant data.

---

3. Row Filtering

Why? To analyze specific groups or remove irrelevant entries.

Example: Filter employees with salary above 60,000

```
df[df['Salary'] > 60000]
```

Combine filters:

```
df[(df['Salary'] > 60000) & (df['Department'] == 'IT')]
```

Filtering lets you zoom into meaningful segments (e.g., high earners, specific departments).

---

4. Sorting Data

Why? Sorting helps rank or prioritize based on values.

```
df.sort_values(by='Experience', ascending=False)
```

Tip: Combine with `head()` to get top performers or outliers.

---

5. Summarizing Data

Why? Summarization gives insights into distributions, totals, or averages.

```
df['Salary'].mean()
df.groupby('Department')['Salary'].mean()
```

Use `.groupby()` for department-level aggregations or comparisons.

---

6. Basic Cleaning

Rename Columns Why? Column names might be messy, inconsistent, or unclear.

```
df.rename(columns={'Experience': 'YearsExperience'}, inplace=True)
```

Handle Missing Values Why? Missing data can break functions or skew results.

Check missing:

```
df.isnull().sum()
```

Remove rows with missing values:

```
df.dropna(inplace=True)
```

Or fill missing values:

```
df['Experience'] = df['Experience'].fillna(0)
```

Combined Demo: Real Use Case

```
import pandas as pd

# Load dataset
df = pd.read_csv("employees.csv")

# Step 1: View structure
print(df.info())

# Step 2: Select relevant columns
df = df[['Name', 'Department', 'Salary', 'Experience']]

# Step 3: Filter employees with 5+ years experience
experienced = df[df['Experience'] >= 5]

# Step 4: Sort by salary
experienced_sorted = experienced.sort_values(by='Salary', ascending=False)

# Step 5: Add new column for tax estimate (just for fun)
df['TaxEstimate'] = df['Salary'] * 0.25

# Step 6: Rename column
df.rename(columns={'Experience': 'YearsExperience'}, inplace=True)

# Step 7: Fill missing experience
df['YearsExperience'] = df['YearsExperience'].fillna(0)

print(df.head())
```

Bonus Dataset for Practice: `students_scores.csv`

(Same as used previously — use it for practice with academic data.)

Summary: Why These Skills Matter

Skill	Purpose
Column selection	Focus analysis on relevant variables
Row filtering	Target groups or exclude unwanted data
Sorting	Rank values or find top/bottom entries

Skill	Purpose
Summarization	Understand data trends and compute metrics
Cleaning	Prepare clean data for analysis or modeling

V.Homework

V. Homework Assignment: Exploring a CSV Dataset with Pandas

Task

Load a dataset using **Pandas** and perform **basic data exploration and cleaning**.

Dataset: `employee_data.csv`

This dataset contains fictional employee records. Here's a preview:

```
ID,Name,Department,Gender,Age,Salary,Experience,Location
101,Alice,HR,Female,29,52000,4,New York
102,Bob,Finance,Male,34,61000,7,Chicago
103,Charlie,IT,Male,41,72000,10,San Francisco
104,Daisy,Finance,Female,28,59000,,Boston
105,Edward,IT,Male,36,73000,9,Chicago
106,Fiona,HR,Female,26,51000,3,New York
107,George,Marketing,Male,30,56000,5,Los Angeles
108,Helen,Finance,Female,38,64000,8,Chicago
109,Ian,Marketing,Male,45,58000,12,Los Angeles
110,Jane,IT,Female,32,70000,6,San Francisco
```

Save this content to a file named `employee_data.csv`.

Homework Steps

1. Load the dataset using `pd.read_csv()`
2. Display the first 5 rows with `.head()`
3. Show structure using `.info()` and statistics with `.describe()`
4. Select and print specific columns: `'Name'`, `'Department'`, `'Salary'`
5. Filter employees with **salary $>$ 60,000** and **experience 5 years**
6. Fill missing values in the `Experience` column with `0`
7. Save the cleaned DataFrame to a new file: `employee_data_cleaned.csv`

Solution Code (with steps)

```
import pandas as pd

# 1. Load the dataset
df = pd.read_csv("employee_data.csv")
```

```python
# 2. Display the first few rows
print("First 5 rows:")
print(df.head())

# 3. Show structure and summary statistics
print("\nDataFrame Info:")
print(df.info())

print("\nDescriptive Statistics:")
print(df.describe())

# 4. Select specific columns
print("\nSelected Columns:")
print(df[['Name', 'Department', 'Salary']])

# 5. Filter based on conditions
filtered_df = df[(df['Salary'] > 60000) & (df['Experience'] >= 5)]
print("\nFiltered Employees (Salary > 60000 and Experience >= 5):")
print(filtered_df)

# 6. Fill missing values
df['Experience'] = df['Experience'].fillna(0)

# 7. Save cleaned dataset
df.to_csv("employee_data_cleaned.csv", index=False)
print("\nCleaned dataset saved to 'employee_data_cleaned.csv'")
```

Lesson 11

Exercise

LESSON 11: MANIPULATING DATA IN PYTHON – PART 1

Hands-On Practice Assignments (20 Exercises)

- Data loading, cleaning, transformation
- Conditionals, loops, user-defined functions
- Performance comparison: Pandas vs. Polars
- NumPy vectorized operations

Dataset: `employee_performance.csv`

```
EmployeeID,Name,Department,Gender,Age,Salary,YearsExperience,PerformanceScore,BonusEligible,Lo
101,Alice,HR,Female,29,52000,4,88,Yes,New York
102,Bob,Finance,Male,34,61000,7,77,No,Chicago
103,Charlie,IT,Male,41,72000,10,90,Yes,San Francisco
```

```
104,Daisy,Finance,Female,28,59000,3,81,Yes,Boston
105,Edward,IT,Male,36,73000,9,85,Yes,Chicago
106,Fiona,HR,Female,26,51000,2,72,No,New York
107,George,Marketing,Male,30,56000,5,75,No,Los Angeles
108,Helen,Finance,Female,38,64000,8,93,Yes,Chicago
109,Ian,Marketing,Male,45,58000,12,65,No,Los Angeles
110,Jane,IT,Female,32,70000,6,91,Yes,San Francisco
```

Save this as `employee_performance.csv` in your project folder.

Part-A. Data Exploration & Cleansing

---

1. Load and Inspect the Dataset **Outcome:** Use Pandas to load and display the first 5 rows.

```
df = pd.read_csv("employee_performance.csv")
print(df.head())
```

---

2. Dataset Overview **Outcome:** Use `.info()`, `.describe()`, and `.shape` to inspect structure and stats.

---

3. Fill Missing Values **Outcome:** Replace any missing experience or salary values (if added later) with the mean.

---

4. Rename Columns for Clarity **Outcome:** Rename `'YearsExperience'` to `'ExperienceYears'`, `'Salary'` to `'AnnualSalary'`.

Part-B. Data Selection, Filtering, Sorting

---

5. Filter High Performers **Outcome:** Filter all employees with `PerformanceScore > 85` and print their names and departments.

---

6. Sort by Experience and Performance **Outcome:** Sort the dataset by `'ExperienceYears'` descending, then `'PerformanceScore'`.

---

7. Conditional Column Creation **Outcome:** Add a column `'PerformanceTier'`:

- `'High'` if score 85
- `'Medium'` if 70–84
- `'Low'` otherwise

Part-C. Aggregation & Grouping

---

8. Department Performance Summary **Outcome:** Group by `'Department'`, calculate:

- Average salary
- Average performance

---

9. Bonus Eligibility Rate by Department **Outcome:** Count how many in each department are `'BonusEligible' == "Yes"` and calculate percentage.

Part-D. NumPy Integrations

---

10. Calculate Z-Scores for Salary **Outcome:** Use NumPy to add a `'SalaryZScore'` column:

```
from scipy.stats import zscore
df['SalaryZScore'] = zscore(df['AnnualSalary'])
```

---

11. Normalize Performance Scores (0 to 1) **Outcome:** Apply min-max normalization using NumPy:

```
df['PerfNorm'] = (df['PerformanceScore'] - df['PerformanceScore'].min()) / (df['PerformanceSco
```

Part-E. Functions & Loops

---

12. Define Function to Flag Promotions **Outcome:** Write a function `is_promotable(score, experience)` returning `"Yes"` if score > 85 and experience > 5.

Apply it across the DataFrame to create a new column.

---

13. Loop Through and Print Custom Reports **Outcome:** Use a `for` loop to print:

"Alice (HR) is eligible for a bonus." "Bob (Finance) is not eligible…"

Part-F. Advanced Filtering and Logic

---

14. Filter by Multiple Conditions **Outcome:** Display IT employees in San Francisco with salary > 70,000.

---

15. Employees Close to Retirement **Outcome:** Add a column `'RetirementRisk'` set to `"Yes"` if `Age > 40`, otherwise `"No"`.

Part-G. Polars Comparisons

---

16. Load Dataset with Polars **Outcome:** Load the same CSV with Polars and display first 5 rows.

```
import polars as pl
pl_df = pl.read_csv("employee_performance.csv")
print(pl_df.head())
```

---

17. Filter in Polars (PerformanceScore > 85) **Outcome:** Use Polars syntax to filter high performers.

```
pl_df.filter(pl.col("PerformanceScore") > 85)
```

---

18. Group and Aggregate in Polars **Outcome:** Group by `'Department'`, calculate mean `Salary` and `PerformanceScore`.

---

19. Add Derived Column in Polars **Outcome:** Add a column `'BonusAmount' = Salary * 0.1` in Polars.

---

20. Compare Pandas vs. Polars Performance ** Outcome:** Use a large version of the dataset (100k+ rows).

- Compare time to filter high performers in Pandas vs. Polars using `time` or `%%timeit`.

Additional:

- Try replicating one Polars operation in Pandas and compare line-by-line syntax.
- Export filtered data to CSV using both libraries.

Lesson 11- Python Data Manipulationi (Solutions)

Use employee_performance.csv

LESSON 11: SOLUTIONS

Part-A. Data Exploration & Cleansing

1. Load and Inspect the Dataset

```
import pandas as pd
df = pd.read_csv("employee_performance.csv")
print(df.head())
```

2. Dataset Overview

```
print(df.info())
print(df.describe())
print(df.shape)
```

3. Fill Missing Values (if any)

```
df['YearsExperience'].fillna(df['YearsExperience'].mean(), inplace=True)
df['Salary'].fillna(df['Salary'].mean(), inplace=True)
```

4. Rename Columns

```
df.rename(columns={'YearsExperience': 'ExperienceYears', 'Salary': 'AnnualSalary'}, inplace=Tru
```

Part-B. Selection, Filtering, Sorting

5. Filter High Performers

```
print(df[df['PerformanceScore'] > 85][['Name', 'Department']])
```

6. Sort by Experience and Performance

```
print(df.sort_values(by=['ExperienceYears', 'PerformanceScore'], ascending=False))
```

7. Performance Tier Classification

```
def tier(score):
    if score >= 85:
        return 'High'
    elif score >= 70:
        return 'Medium'
    else:
        return 'Low'

df['PerformanceTier'] = df['PerformanceScore'].apply(tier)
```

Part-C. Grouping and Aggregation

8. Department Performance Summary

```
print(df.groupby('Department')[['AnnualSalary', 'PerformanceScore']].mean())
```

9. Bonus Eligibility Rate by Department

```
print(df[df['BonusEligible'] == 'Yes'].groupby('Department')['BonusEligible'].count() / df.grou
```

Part-D. NumPy Integrations

10. Salary Z-Scores

```
from scipy.stats import zscore
df['SalaryZScore'] = zscore(df['AnnualSalary'])
```

11. Normalize Performance Scores

```
df['PerfNorm'] = (df['PerformanceScore'] - df['PerformanceScore'].min()) / (df['PerformanceSco
```

Part-E. Functions & Loops

12. Promotion Eligibility Function

```
def is_promotable(score, exp):
    return "Yes" if score > 85 and exp > 5 else "No"

df['Promotable'] = df.apply(lambda row: is_promotable(row['PerformanceScore'], row['Experience
```

13. Loop Through for Bonus Reports

```
for _, row in df.iterrows():
    status = "eligible" if row['BonusEligible'] == "Yes" else "not eligible"
    print(f"{row['Name']} ({row['Department']}) is {status} for a bonus.")
```

Part-F. Advanced Filtering

14. IT Employees in SF with Salary > 70K

```
print(df[(df['Department'] == 'IT') & (df['Location'] == 'San Francisco') & (df['AnnualSalary']
```

15. Retirement Risk Flag

```
df['RetirementRisk'] = df['Age'].apply(lambda age: 'Yes' if age > 40 else 'No')
```

> Polars is not supported in this current environment, but the code below can be run **locally** after installing Polars using:

```
pip install polars
```

Part-G. Polars Solutions (Assignments 16–20)

---

16. Load Dataset with Polars

```
import polars as pl

pl_df = pl.read_csv("employee_performance.csv")
print(pl_df.head())
```

---

17. Filter High Performers in Polars

```
high_perf = pl_df.filter(pl.col("PerformanceScore") > 85)
print(high_perf)
```

---

18. Group and Aggregate in Polars

```
grouped = pl_df.groupby("Department").agg([
    pl.col("Salary").mean().alias("AvgSalary"),
    pl.col("PerformanceScore").mean().alias("AvgPerformance")
])
print(grouped)
```

---

19. Add Derived Column in Polars

```
pl_df = pl_df.with_columns(
    (pl.col("Salary") * 0.10).alias("BonusAmount")
)
print(pl_df.select(["Name", "Salary", "BonusAmount"]))
```

---

20. Compare Pandas vs Polars Performance

Outcome: Use `%%timeit` in Jupyter or Python's `time` module to measure execution time for a filter.

Pandas

```
import pandas as pd
import time

df = pd.read_csv("employee_performance.csv")

start = time.time()
filtered = df[(df['PerformanceScore'] > 85) & (df['Salary'] > 60000)]
end = time.time()
print("Pandas filter time:", end - start)
```

Polars

```
import polars as pl
import time

pl_df = pl.read_csv("employee_performance.csv")

start = time.time()
filtered = pl_df.filter(
    (pl.col("PerformanceScore") > 85) & (pl.col("Salary") > 60000)
)
end = time.time()
print("Polars filter time:", end - start)
```

Lesson 12

I.Data Cleaning With Pandas

I. Data Cleaning with Pandas

Data cleaning is a critical step in any data analysis or machine learning workflow. It involves detecting and correcting (or removing) corrupt or inaccurate records, missing values, duplicate

data, and outliers in a dataset.

Get Sample data

```
import pandas as pd
import numpy as np

data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva', 'Frank', 'Grace', 'Henry', 'Ivy', 'Ali
    'Age': [25, np.nan, 35, 45, np.nan, 29, 33, 22, 39, 25],
    'Salary': [50000, 60000, 70000, 80000, 90000, 1000000, 65000, np.nan, 72000, 50000],
    'Gender': ['F', 'M', 'M', 'M', 'F', np.nan, 'F', 'M', 'F', 'F'],
    'Email': [
        'alice@example.com', 'bob@example.com', 'charlie@example.com', 'david@example.com',
        'eva@example.com', 'frank@example.com', 'grace@example.com', 'henry@example.com',
        'ivy@example.com', 'alice@example.com'  # duplicate email
    ],
    'Country': ['USA', np.nan, np.nan, np.nan, np.nan, 'Slovenia', 'Slovakia', 'Slovenianstan'
}

df = pd.DataFrame(data)
print(df)
```

and optional, if you want to save it to CSV:

```
df.to_csv('sample_dirty_data.csv', index=False)
```

Handling Missing Data

Missing values can significantly affect the quality of insights or model performance. Pandas provides intuitive functions to detect, remove, or impute missing values.

** Detect Missing Values**

```
df.isnull().sum()
```

This returns the number of missing (NaN) entries for each column.

```
print(df.isnull().sum())
```

**** Drop Missing Values**** Drop rows or columns with missing values:

```
df.dropna(inplace=True)  # Drops any rows with at least one NaN
```

Optional parameters:

- `axis=1` – drop columns instead of rows.
- `how='all'` – drop only if **all** values are missing.
- `subset=['Col1', 'Col2']` – check specific columns only.

```
df.dropna(subset=['Age', 'Salary'], how='any', inplace=True)
```

** Fill Missing Values (Imputation)** Impute missing values with a fixed value or a statistic (mean, median, mode):

```
df['Age'].fillna(df['Age'].mean(), inplace=True)
df['Salary'].fillna(df['Salary'].median(), inplace=True)
```

For categorical data:

```
df['Gender'].fillna(df['Gender'].mode()[0], inplace=True)
```

Fill with a custom value:

```
df['Country'].fillna('Unknown', inplace=True)
```

Forward-fill and back-fill:

```
df.fillna(method='ffill', inplace=True)  # forward fill
df.fillna(method='bfill', inplace=True)  # backward fill
```

> **Tip**: Use `SimpleImputer` from `sklearn.impute` for more advanced strategies (like constant, most_frequent, or strategy-specific per column).

Handling Duplicates

Duplicate entries can skew analysis, especially in count-based operations.

Check for Duplicates

```
df.duplicated().sum()
```

Returns the number of duplicate rows (first occurrence is considered unique).

```
df[df.duplicated()]
```

Drop Duplicates

```
df.drop_duplicates(inplace=True)
```

You can also drop duplicates based on specific columns:

```
df.drop_duplicates(subset=['Name', 'Email'], keep='first', inplace=True)
```

> `keep='last'` or `keep=False` to control which duplicates are retained.

Detecting and Handling Outliers

Outliers can distort statistical measures and models. Common methods for detecting them include the IQR method and Z-score.

Using IQR (Interquartile Range)

```
Q1 = df['Salary'].quantile(0.25)
Q3 = df['Salary'].quantile(0.75)
IQR = Q3 - Q1

# Find outliers
outliers = df[(df['Salary'] < Q1 - 1.5 * IQR) | (df['Salary'] > Q3 + 1.5 * IQR)]
```

Remove or Cap/Floor Outliers Remove:

```
df = df[(df['Salary'] >= Q1 - 1.5 * IQR) & (df['Salary'] <= Q3 + 1.5 * IQR)]
```

Cap/floor (Winsorizing):

```
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

df['Salary'] = df['Salary'].clip(lower=lower_bound, upper=upper_bound)
```

Z-Score Method (Standard Score) Use for normally distributed data:

```
from scipy import stats

z_scores = stats.zscore(df['Salary'])
abs_z_scores = np.abs(z_scores)
df = df[abs_z_scores < 3]  # Keep rows with z-score less than 3
```

Tip: Use boxplots to visually identify outliers.

```
import matplotlib.pyplot as plt

plt.boxplot(df['Salary'])
plt.title('Boxplot of Salary')
plt.show()
```

Summary Table

Task	Function Used	Notes
Detect Missing Values	`df.isnull().sum()`	Per column
Drop Missing Values	`df.dropna()`	Use `subset` to target specific columns
Fill Missing Values	`df.fillna()`	Mean/Median/Mode/Custom
Detect Duplicates	`df.duplicated()`	Use `subset` to focus on columns
Drop Duplicates	`df.drop_duplicates()`	Use `keep` to control removal
Detect Outliers (IQR)	`quantile()`	Good for skewed data
Handle Outliers	`clip()` or filter conditionally	Optionally use Z-score

-

Ii. Data Aggregation In Pandas

II. Data Aggregation in Pandas

Data aggregation helps condense large datasets into meaningful summaries by computing metrics like totals, averages, counts, medians, etc. It's especially useful in reporting, dashboards, and group-based analysis.

Data:

```
import pandas as pd

data = {
    'EmployeeID': [101, 102, 103, 104, 105, 106, 107],
    'Name': ['alice johnson', 'Bob Smith', 'CHARLIE Brown', 'david lee', 'Eve Stone', 'Frank Ki
    'Email': [
        'alice@example.com', 'bob@example.org', 'charlie@example.net',
        'david@workplace.com', 'eve@example.com', 'frank@company.org', 'grace@office.net'
    ],
    'Department': ['HR', 'IT', 'IT', 'Sales', 'HR', 'Sales', 'IT'],
    'Location': ['NY', 'NY', 'SF', 'NY', 'SF', 'SF', 'NY'],
    'HireDate': pd.to_datetime(['2020-01-15', '2019-07-10', '2021-03-20', '2022-11-05', '2018-0
    'Salary': [52000, 75000, 82000, 58000, 50000, 60000, 79000],
    'ExperienceYears': [2, 5, 6, 3, 7, 1, 4],
    'IsRemote': ['Yes', 'No', 'Yes', 'No', 'Yes', 'No', 'Yes']
}

df = pd.DataFrame(data)
```

Using `groupby()`

The `groupby()` function splits the data into groups based on one or more columns, applies a function (like `mean()` or `sum()`), and combines the result.

Basic Aggregation

```
df.groupby('Department')['Salary'].mean()
```

This shows the average salary per department.

Multiple Aggregations for Multiple Groups

```
df.groupby(['Department', 'Location'])['Salary'].agg(['mean', 'count'])
```

- `mean`: average salary
- `count`: number of employees per group

Custom Named Aggregations

```
df.groupby('Department').agg(
    Avg_Salary=('Salary', 'mean'),
    Max_Salary=('Salary', 'max'),
    Employee_Count=('EmployeeID', 'count')
)
```

This syntax (introduced in Pandas 0.25+) allows renaming results directly.

Using `agg()` for Custom Aggregations

`agg()` lets you apply multiple functions to one or more columns, especially with groupby.

```
df.groupby('Department').agg({
    'Salary': ['mean', 'max', 'min'],
    'ExperienceYears': ['median', 'std']
})
```

The result is a multi-level column index. Use `.reset_index()` and rename columns if needed.

Flattening MultiIndex Columns

```
agg_df = df.groupby('Department').agg({
    'Salary': ['mean', 'max'],
    'ExperienceYears': ['median']
})

# Flatten column names
agg_df.columns = ['_'.join(col) for col in agg_df.columns]
agg_df.reset_index(inplace=True)
```

Using `pivot_table()`

`pivot_table()` is similar to Excel's pivot tables and great for cross-tabulated summaries.

```
df.pivot_table(values='Salary', index='Department', columns='Location', aggfunc='mean')
```

This creates a matrix where:

- Rows = Departments
- Columns = Locations
- Values = Average Salary

Count of Employees per Department/Location

```
df.pivot_table(index='Department', columns='Location', values='EmployeeID', aggfunc='count', fi
```

`fill_value` replaces `NaN` with 0 for better readability.

Multiple Aggregations in Pivot Table

```
df.pivot_table(index='Department', values=['Salary', 'ExperienceYears'], aggfunc={'Salary': 'me
```

Merging and Joining DataFrames

Combining multiple datasets is essential when data is spread across sources or tables.

Some sample data

```
import pandas as pd

# Employee information
df1 = pd.DataFrame({
    'EmployeeID': [101, 102, 103, 104],
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Department': ['HR', 'IT', 'Sales', 'HR']
})

# Performance records
df2 = pd.DataFrame({
    'EmployeeID': [102, 103, 104, 105],
    'PerformanceScore': [88, 92, 79, 85],
    'ReviewDate': ['2024-06-01', '2024-06-01', '2024-06-01', '2024-06-01']
})
```

Basic Merge Example

```
combined_df = pd.merge(df1, df2, on='EmployeeID', how='inner')
```

Different Merge Types

Type	Description
`inner`	Only matching records in both
`left`	All from left, match from right
`right`	All from right, match from left
`outer`	All records, match where possible

Merge on Multiple Keys

```
pd.merge(df1, df2, on=['EmployeeID', 'Date'], how='left')
```

Add Suffixes for Overlapping Columns

```
pd.merge(df1, df2, on='EmployeeID', suffixes=('_Emp', '_Perf'))
```

Demo Dataset for Practice

Assume the following simplified HR data:

```
df = pd.DataFrame({
    'EmployeeID': [1, 2, 3, 4, 5, 6],
    'Department': ['HR', 'IT', 'IT', 'Sales', 'HR', 'Sales'],
    'Location': ['NY', 'NY', 'SF', 'NY', 'SF', 'SF'],
    'Salary': [50000, 75000, 80000, 55000, 52000, 58000],
    'ExperienceYears': [2, 5, 7, 3, 2, 4]
})
```

Performance data:

```
performance = pd.DataFrame({
    'EmployeeID': [1, 2, 3, 5, 6],
    'PerformanceRating': [3, 4, 5, 2, 4]
})
```

Merge + Group + Aggregate

```
merged = pd.merge(df, performance, on='EmployeeID', how='left')

summary = merged.groupby('Department').agg({
    'Salary': 'mean',
    'PerformanceRating': 'mean',
    'EmployeeID': 'count'
}).rename(columns={'Salary': 'AvgSalary', 'PerformanceRating': 'AvgRating', 'EmployeeID': 'Cou
```

Summary Table

Task	Code Sample
Group by one column	`df.groupby('Col')['Target'].mean()`
Group by multiple columns	`df.groupby(['Col1', 'Col2'])['Target'].agg(['mean'])`
Custom multiple aggregations	`df.groupby('Col').agg({'A': 'mean', 'B': 'sum'})`
Pivot table summary	`df.pivot_table(values='X', index='Y', columns='Z')`
Merge datasets	`pd.merge(df1, df2, on='Key', how='left')`

Iii. Working With Date And Time In Pandas

III. Working with Date and Time in Pandas

Handling datetime data is essential for time series analysis, scheduling, and trends over time.

Dataset:

```
import pandas as pd

# Sample Employee Data
```

```
data = {
    'EmployeeID': [101, 102, 103, 104, 105, 106, 107],
    'Name': ['alice johnson', 'Bob Smith', 'CHARLIE Brown', 'david lee', 'Eve Stone', 'Frank K
    'Email': [
        'alice@example.com', 'bob@example.org', 'charlie@example.net',
        'david@workplace.com', 'eve@example.com', 'frank@company.org', 'grace@office.net'
    ],
    'Department': ['HR', 'IT', 'IT', 'Sales', 'HR', 'Sales', 'IT'],
    'Location': ['NY', 'NY', 'SF', 'NY', 'SF', 'SF', 'NY'],
    'HireDate': pd.to_datetime(['2020-01-15', '2019-07-10', '2021-03-20', '2022-11-05', '2018-
    'Salary': [52000, 75000, 82000, 58000, 50000, 60000, 79000],
    'ExperienceYears': [2, 5, 6, 3, 7, 1, 4],
    'IsRemote': ['Yes', 'No', 'Yes', 'No', 'Yes', 'No', 'Yes']
}

df = pd.DataFrame(data)
```

Converting to Datetime

```
df['HireDate'] = pd.to_datetime(df['HireDate'])
```

- Ensures consistent format for time-based operations.
- Converts string or object columns to proper datetime types.

Extracting Date Components

```
df['Year'] = df['HireDate'].dt.year
df['Month'] = df['HireDate'].dt.month
df['Weekday'] = df['HireDate'].dt.day_name()
```

- Useful for grouping and filtering by month, year, day of week, etc.

Filtering by Date

```
df[df['HireDate'] >= '2023-01-01']
```

- Enables selecting rows from a specific time window.

Creating Date Ranges

```
pd.date_range(start='2024-01-01', periods=5, freq='D')
```

- Generate a sequence of dates for simulation or filling data.

  Use `.dt` accessor to pull time parts (like `.hour`, `.dayofyear`, etc.).

Iv. Working With Strings In Pandas

IV. Working with Strings in Pandas

String processing is a fundamental part of data cleaning, feature extraction, and formatting—especially for columns like names, emails, text notes, and categorical labels.

To apply string functions in Pandas, use the `.str` accessor: `df['column'].str.method()`.

---

1. Basic String Operations

Let's clean and extract some useful information from the `Name` and `Email` columns.

Capitalize names properly

```
df['Name_Clean'] = df['Name'].str.title()
```

Converts `alice johnson` to `Alice Johnson`.

Extract email domain

```
df['EmailDomain'] = df['Email'].str.split('@').str[1]
```

Extracts `"example.com"` from `"alice@example.com"`.

Remove or replace values

```
df['IsRemote_Recode'] = df['IsRemote'].str.replace('Yes', 'Remote').str.replace('No', 'Onsite'
```

---

2. Filtering with String Conditions

These operations are useful for selecting rows based on string patterns.

Check if name contains a word

```
df[df['Name'].str.contains('li', case=False, na=False)]
```

Finds names containing "li" (like `Alice`, `Charlie`, `Grace Li`).

Emails ending with `.org`

```
df[df['Email'].str.endswith('.org', na=False)]
```

Useful for filtering domains (e.g., `.edu`, `.com`, `.gov`).

String length

```
df['NameLength'] = df['Name'].str.len()
```

Counts the number of characters in each name.

---

3. Cleaning and Formatting

String functions help clean up inconsistencies.

Convert to lowercase or uppercase

```
df['Name_Lower'] = df['Name'].str.lower()
df['Name_Upper'] = df['Name'].str.upper()
```

Strip extra whitespace

```
df['Name_Strip'] = df['Name'].str.strip()
```

Removes leading/trailing whitespace.

---

4. Extract and Split

Extract username from email

```
df['EmailUser'] = df['Email'].str.extract(r'(^[^@]+)')
```

Split full name into first and last

```
df[['FirstName', 'LastName']] = df['Name'].str.split(' ', n=1, expand=True)
```

Uses `n=1` to split only on the first space.

---

5. Advanced Matching (Optional)

Startswith / endswith

```
df[df['Name'].str.startswith('A', na=False)]
df[df['Email'].str.endswith('.net', na=False)]
```

Regex search

```
df[df['Email'].str.contains(r'\.org$', regex=True, na=False)]
```

Summary: Common `.str` Operations

Task	Code Example
Capitalize Names	`df['Name'].str.title()`
Extract Email Domain	`df['Email'].str.split('@').str[1]`
Replace Text	`df['IsRemote'].str.replace('Yes', 'Remote')`
Contains Pattern	`df['Name'].str.contains('john', case=False, na=False)`
Ends With `.org`	`df['Email'].str.endswith('.org', na=False)`
String Length	`df['Name'].str.len()`
Split Full Name	`df['Name'].str.split(' ', n=1, expand=True)`
Extract Username	`df['Email'].str.extract(r'(^[^@]+)')`
Strip Whitespace	`df['Name'].str.strip()`
Count Domain Types	`df['Email'].str.extract(r'\.([a-z]+)$')`

V. Working With Index And Multiindex

V. Working with Index and MultiIndex

The index in a DataFrame helps uniquely identify rows and improves performance for lookups and joins. It helps in **data alignment, faster lookups, slicing**, and organizing complex structures like **hierarchical indexes (MultiIndex)**.

Setting and Resetting Index

By default, Pandas uses integer index (0, 1, 2…). We can set meaningful columns as index.

Set `EmployeeID` as index

```
df_indexed = df.set_index('EmployeeID')
print(df_indexed.head())
```

This makes `EmployeeID` the unique identifier for each row.

Reset to default index

```
df_reset = df_indexed.reset_index()
```

> Use `inplace=True` to modify the DataFrame directly.

Setting a MultiIndex

A **MultiIndex** is a hierarchical index — useful when grouping by multiple dimensions like `Department` and `Location`.

Set MultiIndex with `Department` and `Location`

```
df_multi = df.set_index(['Department', 'Location'])
print(df_multi.head())
```

This nests `Location` under each `Department`.

Accessing Data with MultiIndex

Once you have a MultiIndex, you can **select** data by a combination of levels:

```
# Get all rows in the IT department and NY location
print(df_multi.loc[('IT', 'NY')])
```

You can also **slice** across levels using `.loc` with `pd.IndexSlice`:

```
idx = pd.IndexSlice
print(df_multi.loc[idx[:, 'NY'], :])  # All departments in NY
```

Sorting and Working with MultiIndex

A MultiIndex must be **sorted** for slicing and some operations:

```
df_multi_sorted = df_multi.sort_index()
```

Resetting a MultiIndex

If you want to convert a MultiIndex back to regular columns:

```
df_reset_multi = df_multi.reset_index()
```

Example: Aggregating with MultiIndex

Group by Department and Location:

```
grouped = df.groupby(['Department', 'Location'])[['Salary', 'ExperienceYears']].mean()
print(grouped)
```

Access one group:

```
grouped.loc[('IT', 'NY')]
```

Flatten MultiIndex columns after `agg()`:

```
agg_df = df.groupby(['Department', 'Location']).agg({
    'Salary': ['mean', 'max'],
    'ExperienceYears': 'median'
})

# Flatten the columns
agg_df.columns = ['_'.join(col) for col in agg_df.columns]
agg_df.reset_index(inplace=True)
print(agg_df)
```

Flattening MultiIndex Columns

```
df.columns = ['_'.join(col).strip() if isinstance(col, tuple) else col for col in df.columns]
```

- Useful after aggregating with `groupby().agg()`.

Vi. Working With Data Types And Basic Windowing Function

VI. Working with Data Types and basic windowing function

Understanding data types is important for choosing proper preprocessing methods.

Categorical Data

```
df['Department'] = df['Department'].astype('category')
```

- Saves memory and improves performance.
- Categories can be sorted or reordered with `.cat`.

Continuous Data

These are typically **numerical values** that can take many values within a range.

```
df['Salary'].describe()
```

- Normalize or standardize:

```
df['Salary_z'] = (df['Salary'] - df['Salary'].mean()) / df['Salary'].std()
```

Binary Data

```
df['IsRemote'].unique()  # e.g., ['Yes', 'No'] or [1, 0]
```

```
df['IsRemote'] = df['IsRemote'].map({'Yes': 1, 'No': 0})
```

- Binary columns can be used for classification and filtering.

Basic Windowing Functions

Window functions compute statistics over a sliding or cumulative window.

Rolling Windows

```
df['RollingAvg'] = df['Salary'].rolling(window=3).mean()
```

- Good for smoothing out time series trends.
- Use `.std()`, `.sum()`, etc., as well.

Expanding Window

```
df['CumulativeSum'] = df['Salary'].expanding().sum()
```

- Keeps growing the window to include all previous rows.

Cumulative Functions

```
df['SalaryCumSum'] = df['Salary'].cumsum()
df['SalaryCumMax'] = df['Salary'].cummax()
```

- Ideal for running totals, maximums, or minimums over time.

Ranking and Percentiles

```
df['Rank'] = df['Salary'].rank()
df['Percentile'] = df['Salary'].rank(pct=True)
```

Vii.Hands-On Practice

VII. Hands-on Practice with `employee_performance.csv`

Practice Tasks

1. Load dataset and check for missing values
2. Fill missing `YearsExperience` with mean
3. Remove any duplicated rows
4. Identify outliers in `PerformanceScore`
5. Group by `Department` to calculate average salary and experience
6. Create a pivot table showing average salary per department/location
7. Merge with another mini dataset (e.g., project assignments)

Step 1: Load dataset and check for missing values

```
import pandas as pd

# Load the dataset
df = pd.read_csv("employee_performance.csv")

# Display basic info
print(df.info())

# Check for missing values
print(df.isnull().sum())
```

Step 2: Fill missing `YearsExperience` with mean

```
mean_experience = df['YearsExperience'].mean()
df['YearsExperience'].fillna(mean_experience, inplace=True)

print("Missing values after imputation:\n", df.isnull().sum())
```

Step 3: Remove duplicated rows

```
# Check for duplicates
print("Duplicates found:", df.duplicated().sum())

# Drop duplicates
df.drop_duplicates(inplace=True)

# Confirm
print("Remaining duplicates:", df.duplicated().sum())
```

Step 4: Identify outliers in `PerformanceScore` using IQR

```
Q1 = df['PerformanceScore'].quantile(0.25)
Q3 = df['PerformanceScore'].quantile(0.75)
IQR = Q3 - Q1

outliers = df[(df['PerformanceScore'] < Q1 - 1.5 * IQR) | (df['PerformanceScore'] > Q3 + 1.5 *

print("Outliers detected:\n", outliers)
```

Optionally remove or clip outliers:

```
df['PerformanceScore'] = df['PerformanceScore'].clip(lower=Q1 - 1.5 * IQR, upper=Q3 + 1.5 * IQ
```

Step 5: Group by `Department` to calculate average salary and experience

```
department_summary = df.groupby('Department').agg({
    'Salary': 'mean',
    'YearsExperience': 'mean'
}).rename(columns={
    'Salary': 'AvgSalary',
    'YearsExperience': 'AvgExperience'
})

print(department_summary)
```

Step 6: Create a pivot table of average salary per department/location

```
pivot_salary = pd.pivot_table(
    df,
    values='Salary',
    index='Department',
    columns='Location',
    aggfunc='mean',
    fill_value=0
)

print(pivot_salary)
```

Step 7: Merge with another dataset: `project_assignments.csv`

```
# Example second dataset
projects = pd.DataFrame({
    'EmployeeID': [101, 102, 103, 104, 105],
    'Project': ['Apollo', 'Zeus', 'Hermes', 'Athena', 'Poseidon']
})
```

```
# Merge datasets
df_merged = pd.merge(df, projects, on='EmployeeID', how='left')

print(df_merged[['EmployeeID', 'Name', 'Project']])
```

Optional Final Step: Save the Cleaned Data

```
df_merged.to_csv("cleaned_employee_performance.csv", index=False)
```

Outline

** LESSON 12: MANIPULATING DATA IN PYTHON – PART 2**

DATA AGGREGATION AND CLEANING

** Goal:** Equip learners with essential techniques to clean, prepare, and summarize data using **Pandas** (and optionally Polars).

Recap of Lesson 11

- Reviewed Pandas & NumPy basics
- Practiced selection, filtering, sorting
- Created functions, used loops, and compared Polars vs. Pandas
- Focused on performance insights and transformation pipelines

---

1. Data Cleaning in Pandas

Data cleaning ensures accuracy and consistency before analysis or modeling.

Handling Missing Data

Detect missing values

```
df.isnull().sum()
```

Drop missing values

```
df.dropna(inplace=True)
```

Fill missing values

```
df['Column'].fillna(df['Column'].mean(), inplace=True)
```

Tip: Use `median()` or `mode()` for categorical data.

Handling Duplicates

Check duplicates

```
df.duplicated().sum()
```

Drop duplicates

```
df.drop_duplicates(inplace=True)
```

Detecting and Handling Outliers

Using IQR method

```
Q1 = df['Salary'].quantile(0.25)
Q3 = df['Salary'].quantile(0.75)
IQR = Q3 - Q1
outliers = df[(df['Salary'] < Q1 - 1.5 * IQR) | (df['Salary'] > Q3 + 1.5 * IQR)]
```

Optionally: Remove or cap/floor them using `clip()`.

---

2. Data Aggregation in Pandas

Aggregating data helps summarize information at group level (e.g., by department, location).

Using `groupby()`

```
df.groupby('Department')['Salary'].mean()
```

Group by multiple columns:

```
df.groupby(['Department', 'Location'])['Salary'].agg(['mean', 'count'])
```

Using `agg()` for custom aggregations

```
df.groupby('Department').agg({
    'Salary': ['mean', 'max'],
    'ExperienceYears': 'median'
})
```

Using `pivot_table()`

```
df.pivot_table(values='Salary', index='Department', columns='Location', aggfunc='mean')
```

Pivot tables are great for cross-tab reports.

Merging and Joining DataFrames

Combine multiple datasets for integrated analysis.

Example: Merge employee and performance tables

```
pd.merge(df1, df2, on='EmployeeID', how='inner')
```

- `how='inner'`, `'outer'`, `'left'`, `'right'`

---

3. Hands-on Practice

Dataset: `employee_performance.csv`

Practice Tasks

1. Load dataset and check for missing values
2. Fill missing `YearsExperience` with mean
3. Remove any duplicated rows
4. Identify outliers in `PerformanceScore`
5. Group by `Department` to calculate average salary and experience
6. Create a pivot table showing average salary per department/location
7. Merge with another mini dataset (e.g., project assignments)

---

4. Homework Assignment

Clean and summarize a dataset using Pandas

1. Load a dataset (`employee_performance.csv`)
2. Identify and fill missing values
3. Drop duplicate records (if any)
4. Detect and optionally handle outliers in salary
5. Group by department and calculate:
   - Average salary
   - Headcount
6. Create a pivot table of average performance score by department/location
7. Export the cleaned dataset to a new CSV

Additional Topics to Cover in Lesson 12

Pandas

1. Working with Dates and Time

- `pd.to_datetime()`
- Extracting year, month, day: `df['date'].dt.year`
- Filtering by date ranges

Example:

```
df['HireDate'] = pd.to_datetime(df['HireDate'])
df[df['HireDate'].dt.year > 2020]
```

---

2. Value Counts and Unique Values

- `.value_counts()` for frequency
- `.nunique()` for distinct counts

Example:

```
df['Department'].value_counts()
df['Location'].nunique()
```

---

3. String Operations

- `.str.lower()`, `.str.contains()`, `.str.replace()`

Example:

```
df[df['Name'].str.contains("an")]
df['Name'] = df['Name'].str.upper()
```

---

4. Cutting and Binning Data

- Use `pd.cut()` or `pd.qcut()` to segment continuous values into categories

Example:

```
df['ExperienceLevel'] = pd.cut(df['ExperienceYears'], bins=[0, 3, 6, 10], labels=["Junior", "M
```

NumPy

5. Array Broadcasting and Element-wise Operations

- Apply transformations over arrays efficiently

Example:

```
salaries = df['Salary'].values
adjusted = salaries * 1.1  # apply 10% raise
```

---

6. Logical Indexing and Masking with NumPy

- Use `np.where()` or boolean masks

Example:

```
df['Bonus'] = np.where(df['PerformanceScore'] > 85, 5000, 0)
```

Polars

7. Lazy Execution in Polars

- Use `scan_csv()` and `.collect()` for efficient data pipelines

Example:

```
lazy_df = pl.scan_csv("employee_performance.csv")
result = lazy_df.filter(pl.col("Salary") > 60000).select(["Name", "Salary"]).collect()
```

---

8. Apply/Map-Like Transformations (Functional Style)

- Polars discourages row-wise loops; use expressions and chaining

Example:

```
pl_df.with_columns(
    (pl.col("PerformanceScore") > 85).cast(pl.Utf8).alias("HighPerformer")
)
```

Bonus Concepts (Optional)

9. Chaining in Pandas (Method Chaining Style) Encourage writing clean, chainable data transformation pipelines:

```
df_filtered = (
    df[df['Salary'] > 60000]
    .groupby('Department')
    .agg({'Salary': 'mean'})
    .reset_index()
)
```

Suggested Integration into Lesson 12:

Topic	Integration Point
Value counts / string ops	During data exploration
Grouping & pivoting	Include `cut()`, `qcut()` bins
NumPy masks, where	While building bonus/salary logic

Topic	Integration Point
Polars lazy execution	At end or bonus material
Date parsing	Add optional column in dataset

Lesson 13

Exercise

LESSON 13: Exercise

Dataset: `sales_data.csv`

OrderID	Customer	Region	Product	Quantity	UnitPrice	OrderDate	SalesRep
1001	Alice	East	Laptop	2	900	2023-01-15	John Doe
1002	Bob	West	Monitor	5	200	2023-01-18	Jane Smith
1003	Charlie	East	Keyboard	3	50	2023-02-05	John Doe
1004	Alice	North	Laptop	1	950	2023-03-01	Amy Adams
1005	David	South	Mouse	10	25	2023-03-10	Jane Smith
1006	Eve	West	Monitor	2	210	2023-04-02	John Doe
1007	Frank	East	Laptop	1	1000	2023-04-10	Amy Adams
1008	Grace	North	Mouse	6	30	2023-04-22	Jane Smith
1009	Heidi	South	Keyboard	4	60	2023-05-03	John Doe
1010	Ivan	West	Monitor	3	205	2023-05-15	Amy Adams

```
import pandas as pd

data = [
    {"OrderID": 1001, "Customer": "Alice", "Region": "East", "Product": "Laptop", "Quantity": 2
    {"OrderID": 1002, "Customer": "Bob", "Region": "West", "Product": "Monitor", "Quantity": 5
    {"OrderID": 1003, "Customer": "Charlie", "Region": "East", "Product": "Keyboard", "Quantity
    {"OrderID": 1004, "Customer": "Alice", "Region": "North", "Product": "Laptop", "Quantity":
    {"OrderID": 1005, "Customer": "David", "Region": "South", "Product": "Mouse", "Quantity": 1
    {"OrderID": 1006, "Customer": "Eve", "Region": "West", "Product": "Monitor", "Quantity": 2
    {"OrderID": 1007, "Customer": "Frank", "Region": "East", "Product": "Laptop", "Quantity": 1
    {"OrderID": 1008, "Customer": "Grace", "Region": "North", "Product": "Mouse", "Quantity": 6
    {"OrderID": 1009, "Customer": "Heidi", "Region": "South", "Product": "Keyboard", "Quantity"
    {"OrderID": 1010, "Customer": "Ivan", "Region": "West", "Product": "Monitor", "Quantity": 3
]

df = pd.DataFrame(data)
```

Assignments

1. **Load the dataset into a pandas DataFrame.**
2. **Display the first 5 rows and data types of all columns.**
3. **Create a new column `TotalPrice` as Quantity × UnitPrice.**
4. **Filter rows where the Region is 'East' and display them.**
5. **Filter rows where Quantity is more than 3 and UnitPrice is less than 100.**
6. **Sort the dataset by `TotalPrice` in descending order.**
7. **Group data by Region and calculate total Quantity per Region.**
8. **Group data by SalesRep and calculate average UnitPrice.**
9. **Find the top 3 most sold Products (by Quantity).**
10. **Calculate total revenue (sum of TotalPrice) by SalesRep.**
11. **Add a column `OrderMonth` extracted from OrderDate.**
12. **Group data by `OrderMonth` and calculate the monthly total sales.**
13. **Find the Product with the highest UnitPrice.**
14. **Using a lambda function, create a new column `HighValue` (True if TotalPrice > 500).**
15. **Count how many orders were HighValue.**
16. **Display unique values in Region and Product columns.**
17. **Find customers who ordered more than once.**
18. **Replace 'Monitor' with 'LCD Monitor' in the Product column.**
19. **Drop the column `OrderID`.**
20. **Convert all Customer names to uppercase.**
21. **Convert all SalesRep names to lowercase.**
22. **Concatenate Customer and Region into a new column called CustomerRegion.**
23. **Check if the product name contains the word "top" (case-insensitive), and create a boolean column ContainsTop.**
24. **Extract only letters from SalesRep names using regex (remove spaces and non-letters).**
25. **Create a new column OrderDay to extract the day of the month from OrderDate.**
26. **Create a new column OrderYear to extract the year from OrderDate.**
27. **Create a new column FormattedDate that formats the date as "DD-MM-YYYY".**
28. **Filter and display only the orders made after March 1, 2023.**
29. **Create pivot table of Total Sales by Region and Product**
30. **Create pivot table of Average Unit Price by SalesRep and Product**
31. **Print table with number of Orders per Month and Region**
32. **Export the final DataFrame to a new CSV file.**

Lesson 13- Solutions

LESSON 13: Solutions

```
import pandas as pd

# 1. Load dataset
df = pd.read_csv("sales_data.csv")

# 2. Display first 5 rows and data types
print(df.head())
```

```
print(df.dtypes)

# 3. Create TotalPrice
df['TotalPrice'] = df['Quantity'] * df['UnitPrice']

# 4. Filter Region = 'East'
east_df = df[df['Region'] == 'East']

# 5. Quantity > 3 and UnitPrice < 100
filtered_df = df[(df['Quantity'] > 3) & (df['UnitPrice'] < 100)]

# 6. Sort by TotalPrice descending
sorted_df = df.sort_values(by='TotalPrice', ascending=False)

# 7. Group by Region, total Quantity
region_qty = df.groupby('Region')['Quantity'].sum()

# 8. Group by SalesRep, avg UnitPrice
salesrep_avg_price = df.groupby('SalesRep')['UnitPrice'].mean()

# 9. Top 3 most sold Products
top_products = df.groupby('Product')['Quantity'].sum().sort_values(ascending=False).head(3)

# 10. Total revenue by SalesRep
salesrep_revenue = df.groupby('SalesRep')['TotalPrice'].sum()

# 11. Add OrderMonth
df['OrderDate'] = pd.to_datetime(df['OrderDate'])
df['OrderMonth'] = df['OrderDate'].dt.month

# 12. Monthly total sales
monthly_sales = df.groupby('OrderMonth')['TotalPrice'].sum()

# 13. Product with highest UnitPrice
max_price_product = df.loc[df['UnitPrice'].idxmax(), 'Product']

# 14. HighValue column
df['HighValue'] = df['TotalPrice'].apply(lambda x: x > 500)

# 15. Count HighValue orders
high_value_count = df['HighValue'].sum()

# 16. Unique values
unique_regions = df['Region'].unique()
unique_products = df['Product'].unique()

# 17. Customers with > 1 order
multiple_orders = df['Customer'].value_counts()
```

```
repeat_customers = multiple_orders[multiple_orders > 1].index.tolist()

# 18. Replace 'Monitor' with 'LCD Monitor'
df['Product'] = df['Product'].replace('Monitor', 'LCD Monitor')

# 19. Drop OrderID
df.drop(columns=['OrderID'], inplace=True)


#20. Convert all Customer names to uppercase
df["CustomerUpper"] = df["Customer"].str.upper()

#21. Convert all SalesRep names to lowercase
df["SalesRepLower"] = df["SalesRep"].str.lower()

#22. Concatenate Customer and Region into a new column called `CustomerRegion`
df["CustomerRegion"] = df["Customer"] + " - " + df["Region"]


#23. Check if the product name contains the word "top" (case-insensitive), and create a boolea
df["ContainsTop"] = df["Product"].str.contains("top", case=False)


#24. Extract only letters from SalesRep names using regex (remove spaces and non-letters)
df["SalesRepClean"] = df["SalesRep"].str.replace(r"[^a-zA-Z]", "", regex=True)




#25. Create a new column OrderDay to extract the day of the month from OrderDate
df["OrderDay"] = df["OrderDate"].dt.day


#26. Create a new column OrderYear to extract the year from OrderDate
df["OrderYear"] = df["OrderDate"].dt.year


#27. Create a new column `FormattedDate` that formats the date as "DD-MM-YYYY"
df["FormattedDate"] = df["OrderDate"].dt.strftime("%d-%m-%Y")


#28. Filter and display only the orders made after March 1, 2023
filtered_df = df[df["OrderDate"] > "2023-03-01"]
print(filtered_df)


# 29. Create pivot table of Total Sales by Region and Product
df["TotalSales"] = df["Quantity"] * df["UnitPrice"]
```

```python
pivot = pd.pivot_table(df, values="TotalSales", index="Region", columns="Product", aggfunc="sum")
print(pivot)

# 30. Create pivot table of  Average Unit Price by SalesRep and Product
pivot = pd.pivot_table(df, values="UnitPrice", index="SalesRep", columns="Product", aggfunc="mean")
print(pivot)

# 31. Print table with number of Orders per Month and Region
df["Month"] = df["OrderDate"].dt.to_period("M")
pivot = pd.pivot_table(df, values="OrderID", index="Month", columns="Region", aggfunc="count", fill_value=0)
print(pivot)

# 29. Export to CSV
df.to_csv("processed_sales_data.csv", index=False)
```

Lesson 14

0.Concepts

Concepts

1. What is an RDBMS?

RDBMS stands for **Relational Database Management System**. It is a software system used to manage relational databases — databases that store data in **tables** (also called relations).

- Each table consists of **rows** (records) and **columns** (attributes or fields).
- Data in RDBMS is organized to reduce redundancy and ensure data integrity.

Popular RDBMSs include **MySQL**, **PostgreSQL**, **Oracle**, **SQL Server**, and **SQLite**.

---

2. Why Do We Use Tables in RDBMS?

Tables:

- Represent **entities** (like customers, orders, products, etc.).
- Are easy to visualize, manage, and query.
- Help in maintaining structured and related data.

Example:

```
Customers Table:
+----+----------+-----------+
| ID | Name     | Email     |
+----+----------+-----------+
| 1  | Alice    | a@x.com   |
| 2  | Bob      | b@x.com   |
+----+----------+-----------+
```

---

3. What is Normalization? What are Normal Forms?

Normalization is the process of organizing data to reduce redundancy and improve data integrity.

It involves splitting data into multiple tables and defining relationships between them.

Common Normal Forms:

1. **1NF (First Normal Form)**:
   - No repeating groups or arrays.
   - Each field contains only atomic (indivisible) values.
2. **2NF (Second Normal Form)**:
   - Must be in 1NF.
   - No partial dependency (every non-key column depends on the whole primary key).
3. **3NF (Third Normal Form)**:
   - Must be in 2NF.
   - No transitive dependencies (non-key column depends only on the primary key).

Higher forms (like BCNF, 4NF) are more advanced and used in specific scenarios.

---

4. What is a Primary Key?

A **Primary Key** is a column (or set of columns) that uniquely identifies each row in a table.

- Cannot be NULL.
- Must be unique.

Example:

```
Customer(ID int PRIMARY KEY, Name varchar)
```

---

5. What is a Foreign Key?

A **Foreign Key** is a column (or set of columns) in one table that refers to the **Primary Key** in another table.

- It establishes a **relationship** between two tables.
- Maintains **referential integrity**.

Example:

```
Orders Table:
+----+------------+-------------+
| ID | CustomerID | OrderTotal  |
+----+------------+-------------+
| 1  | 1          | 100.00      |
```

```
| 2  | 2          | 150.00      |
+----+------------+-------------+
```

Here, `CustomerID` is a foreign key referring to `Customers(ID)`.

---

6. What is Data Integrity?

Data Integrity ensures that the data in the database is accurate and consistent.

Types:

- **Entity Integrity**: Ensures each table has a primary key and unique rows.
- **Referential Integrity**: Ensures that foreign keys correctly reference primary keys in other tables.
- **Domain Integrity**: Ensures that data entries are valid for the column's data type and constraints (like NOT NULL, CHECK, etc.).

---

7. Why Use Relationships Between Tables?

Relationships allow you to:

- Avoid data duplication (e.g., no need to repeat customer info in every order).
- Maintain consistency (e.g., if a customer is deleted, you know what to do with related orders).
- Query related data easily using **JOINs**.

I.Foundations Of Relational Databases

1. Foundations of Relational Databases

Relational databases are a **foundational technology** in computer science used to store structured data. They organize information into **tables (also called relations)**, where:

- **Rows** represent **individual records**.
- **Columns** represent **attributes** of the data.
- **Primary keys** uniquely identify records in a table.
- **Foreign keys** establish relationships between tables.

This model enables you to **link data across tables**, reducing redundancy and improving consistency.

**** Historical Background:**** The **relational model** was proposed by **Edgar F. Codd** in **1970** in a landmark paper titled *"A Relational Model of Data for Large Shared Data Banks"*. Codd's model provided a mathematical foundation using **set theory** and **predicate logic**, which made it easier to query and manipulate data using a **declarative language** (like SQL). (source: wiki stuff)

Key milestones:

- **1979**: First commercial RDBMS (Oracle v2)
- **1986**: SQL standardized by ANSI
- **1990s–2000s**: Dominance of relational systems like Oracle, MSSQL Server, MySQL, PostgreSQL
- **Today**: RDBMS still dominate, though often complemented by specialized systems (e.g., NoSQL, time-series DBs)

Key Concepts in Relational Databases:

Concept	Description
Table	A collection of rows and columns. Also called a "relation".
Row (Record)	A single entry in a table.
Column (Field)	A data attribute for the record.
Primary Key	A column (or set) that uniquely identifies a row.
Foreign Key	A reference to the primary key in another table.
SQL (Structured Query Language)	The language used to manage and query relational databases.

Popular Relational Database Systems (RDBMS Providers):

Provider	Notable Product	Description
Oracle	Oracle DB	Enterprise-grade, widely used
Microsoft	SQL Server	Windows-integrated, powerful GUI tools
IBM	Db2	Strong in legacy enterprise environments
PostgreSQL	PostgreSQL	Open-source, standards-compliant, robust
MySQL	MySQL, MariaDB	Widely used in web development
SQLite	SQLite	Lightweight, file-based, embedded use

Other Database Paradigms (Beyond Relational):

Type	Examples	Use Case
NoSQL	MongoDB, Couchbase	Unstructured or semi-structured data
Time-Series	InfluxDB, TimescaleDB	Sensor data, logs, time-stamped events
Key-Value	Redis, DynamoDB	Fast lookup of values by key
Document	MongoDB, CouchDB	JSON/BSON structured documents
Graph	Neo4j, ArangoDB	Complex relationships (social networks, graphs)

Creating a Simple Relational Model in SQLite Let's create a **SQLite database** with three connected tables: `Customers`, `Orders`, and `Products`.

```
%CREATE shop.db shopdb

-- Create Customers table
CREATE TABLE Customers (
    CustomerID INTEGER PRIMARY KEY,
    Name TEXT NOT NULL,
    Email TEXT UNIQUE NOT NULL
);

-- Create Products table
CREATE TABLE Products (
    ProductID INTEGER PRIMARY KEY,
    Name TEXT NOT NULL,
    Price REAL NOT NULL
);

-- Create Orders table
CREATE TABLE Orders (
    OrderID INTEGER PRIMARY KEY,
    OrderDate TEXT NOT NULL,
    CustomerID INTEGER NOT NULL,
    ProductID INTEGER NOT NULL,
    Quantity INTEGER NOT NULL,
    FOREIGN KEY(CustomerID) REFERENCES Customers(CustomerID),
    FOREIGN KEY(ProductID) REFERENCES Products(ProductID)
);
```

INSERT Statements:

```
-- Insert Customers
INSERT INTO Customers (Name, Email) VALUES ('Alice Smith', 'alice@example.com');
INSERT INTO Customers (Name, Email) VALUES ('Bob Johnson', 'bob@example.com');

-- Insert Products
INSERT INTO Products (Name, Price) VALUES ('Laptop', 1200.00);
INSERT INTO Products (Name, Price) VALUES ('Headphones', 150.00);

-- Insert Orders
INSERT INTO Orders (OrderDate, CustomerID, ProductID, Quantity)
VALUES ('2025-07-01', 1, 1, 1);  -- Alice buys a Laptop

INSERT INTO Orders (OrderDate, CustomerID, ProductID, Quantity)
VALUES ('2025-07-02', 2, 2, 2);  -- Bob buys two Headphones
```

Example Query: Join Customers and Orders

```
SELECT
    Customers.Name AS Customer,
    Products.Name AS Product,
    Orders.Quantity,
    Orders.OrderDate
FROM Orders
JOIN Customers ON Orders.CustomerID = Customers.CustomerID
JOIN Products ON Orders.ProductID = Products.ProductID;
```

Software needed

https://sqlitebrowser.org/ Visual Studio Code

Ii. Sql Basics For Python Programmers

II. SQL Basics for Python Programmers

**** a. Introduction to CRUD Operations**** **CRUD** stands for:

Operation	SQL Keyword	Purpose
Create	`INSERT`	Add new records to a table
Read	`SELECT`	Retrieve data from a table
Update	`UPDATE`	Modify existing records
Delete	`DELETE`	Remove records from a table

Examples of Each CRUD Operation Assume we have this `Customers` table:

```
CREATE TABLE Customers (
    CustomerID INTEGER PRIMARY KEY,
    Name TEXT NOT NULL,
    Email TEXT UNIQUE NOT NULL
);
```

1. SELECT – Read All Customers:

```
SELECT * FROM Customers;
```

2. INSERT – Add a New Customer:

```
INSERT INTO Customers (Name, Email)
VALUES ('Alice Johnson', 'alice@example.com');
```

3. UPDATE – Modify a Customer's Email:

```
UPDATE Customers
SET Email = 'alice_new@example.com'
WHERE Name = 'Alice Johnson';
```

4. DELETE – Remove a Customer:

```
DELETE FROM Customers
WHERE Name = 'Alice Johnson';
```

Python Integration Tip In Python, you can use the built-in `sqlite3` library:

```
import sqlite3

conn = sqlite3.connect("shop.db")
cur = conn.cursor()

# Insert example
cur.execute("INSERT INTO Customers (Name, Email) VALUES (?, ?)", ("John Doe", "john@example.co
conn.commit()

# Query example
cur.execute("SELECT * FROM Customers")
rows = cur.fetchall()
for row in rows:
    print(row)

conn.close()
```

You can also use **SQLAlchemy** for an object-relational mapping (ORM) approach.

---

b. Overview of SQL Predicates and Clauses

SQL's power comes from its **declarative syntax** and expressive **clauses**. What is this?

WHat does **declarative** and **expressive** mean?

Declarative Syntax (vs. Imperative) In SQL, **you describe *what* you want**, not **how** to get it.

- **Declarative**: SQL is *declarative*, meaning you write a query like:

  ```
  SELECT Name FROM Customers WHERE Email LIKE '%@example.com';
  ```

  This says: "Give me names of customers with emails ending in `@example.com`." You don't tell the database *how* to search, loop, or filter — the database engine figures that out.

- **Imperative**: In contrast, languages like Python or JavaScript are *imperative* — you'd write a loop to manually process data.

Summary: SQL focuses on *what* result you want, not *how* to compute it. The database engine handles the logic.

Expressive Clauses **Clauses** are the building blocks of SQL queries — they let you:

Clause	What it does
`SELECT`	Choose which columns to return
`FROM`	Choose which table(s) to query
`WHERE`	Filter rows based on conditions
`ORDER BY`	Sort the results
`GROUP BY`	Group rows for aggregation
`HAVING`	Filter groups (after `GROUP BY`)
`JOIN`	Combine rows from multiple tables
`LIMIT`	Restrict the number of rows returned

These **clauses make SQL very powerful and readable** — you can construct rich queries with just a few lines of code.

Example: Declarative and Expressive

```sql
SELECT CustomerID, COUNT(*) AS TotalOrders
FROM Orders
WHERE OrderDate >= '2025-01-01'
GROUP BY CustomerID
HAVING COUNT(*) > 5
ORDER BY TotalOrders DESC;
```

This one line of SQL:

- filters orders by date
- groups them by customer
- counts how many orders each customer made
- filters groups to only include those with more than 5 orders
- sorts results by most orders

All without writing a single loop or manual filter!

**** WHERE – Filter Records**** Select all orders made by Customer ID 1:

```sql
SELECT * FROM Orders WHERE CustomerID = 1;
```

**** ORDER BY – Sort Results**** Sort customers alphabetically:

```
SELECT * FROM Customers ORDER BY Name ASC;
```

Sort customers by most recent email update (if timestamp column exists):

```
SELECT * FROM Customers ORDER BY LastUpdated DESC;
```

**** GROUP BY – Aggregate Data**** Find how many orders each customer has made:

```
SELECT CustomerID, COUNT(*) AS TotalOrders
FROM Orders
GROUP BY CustomerID;
```

You can also use `HAVING` to filter aggregated results:

```
SELECT CustomerID, COUNT(*) AS TotalOrders
FROM Orders
GROUP BY CustomerID
HAVING COUNT(*) > 2;
```

Key SQL Concepts

Concept	Description
Clause	A keyword-driven part of a SQL statement (e.g., `WHERE`, `ORDER BY`)
Predicate	A condition used in `WHERE`, `HAVING`, etc., to filter rows
Wildcard	`*` selects all columns, `%` used in `LIKE` for pattern matching
Aggregate Functions	`COUNT()`, `SUM()`, `AVG()`, `MAX()`, `MIN()` for grouping data
Joins	Combine rows from two or more tables based on related columns

Iii. Sql Joins And Relationships

III. SQL Joins and Relationships

What Are Joins? A **JOIN** in SQL lets you **combine rows from two or more tables** based on a **related column**, usually using **foreign keys**. This is essential in **relational databases**, where data is split across multiple tables to avoid redundancy.

Why Use Joins?

- Normalize data (avoid duplication)
- Connect related entities (e.g., Customers Orders)
- Prepare merged data for analysis or visualization
- Keep schema flexible and scalable

Key Join Types (with Visual Explanation)

We'll use two example tables:

Customers Table

CustomerID	Name
1	Alice
2	Bob
3	Charlie

Orders Table

OrderID	CustomerID	OrderDate
101	1	2025-07-01
102	1	2025-07-02
103	2	2025-07-03

INNER JOIN Returns only **matching rows** from both tables.

```
SELECT Customers.Name, Orders.OrderDate
FROM Customers
INNER JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
```

Result:

Name	OrderDate
Alice	2025-07-01
Alice	2025-07-02
Bob	2025-07-03

Diagram:

```
Customers     Orders
   [1]  Alice   [101] CustID:1
   [2]  Bob     [102] CustID:1
   [3]  Charlie [103] CustID:2

         INNER JOIN on CustomerID

      Only IDs present in BOTH → 1, 2
```

Charlie has no orders, so is excluded.

LEFT JOIN (LEFT OUTER JOIN) Returns **all rows from the left table** (Customers), and matched rows from the right table (Orders). If there's no match, the right side shows `NULL`.

```
SELECT Customers.Name, Orders.OrderDate
FROM Customers
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
```

Result:

Name	OrderDate
Alice	2025-07-01
Alice	2025-07-02
Bob	2025-07-03
Charlie	NULL

Diagram:

```
Customers     Orders
   [1] Alice    [101] CustID:1
   [2] Bob      [102] CustID:1
   [3] Charlie  [103] CustID:2

         LEFT JOIN on CustomerID

      All Customers, even if no orders
```

Charlie is included even without matching orders.

(Optional) RIGHT JOIN and FULL JOIN SQLite does **not** support `RIGHT JOIN` or `FULL OUTER JOIN` directly, but they exist in other RDBMS (PostgreSQL, SQL Server, etc.).

- **RIGHT JOIN**: Like LEFT JOIN, but starts from the right table
- **FULL OUTER JOIN**: Combines results from both sides, even if there's no match on either side

---

How This Helps in Python

When you're analyzing data in **Pandas**, `JOINs` are equivalent to **`merge()`**:

```
# Using pandas
import pandas as pd

customers = pd.DataFrame({
    'CustomerID': [1, 2, 3],
    'Name': ['Alice', 'Bob', 'Charlie']
})
```

```
orders = pd.DataFrame({
    'OrderID': [101, 102, 103],
    'CustomerID': [1, 1, 2],
    'OrderDate': ['2025-07-01', '2025-07-02', '2025-07-03']
})

# INNER JOIN
merged_inner = pd.merge(customers, orders, on='CustomerID', how='inner')

# LEFT JOIN
merged_left = pd.merge(customers, orders, on='CustomerID', how='left')
```

Key Concepts Recap

Term	Meaning
JOIN	Combines rows from multiple tables
INNER JOIN	Only matching rows from both tables
LEFT JOIN	All rows from left table, matched rows from right or NULL
RIGHT JOIN	All rows from right table, matched rows from left or NULL
FULL JOIN	All rows from both tables, NULL where no match
Foreign Key	A column in one table that references the primary key in another
Aliasing	Use `AS` to rename columns or tables temporarily in a query

Iv. Hans-On Practice

IV. Hands-On Practice

Objective: Learn by Doing SQL is best learned through **hands-on practice**. In this section, learners will:

- Interact with real data using SQL queries
- Understand joins, filters, and aggregations
- See how SQL connects to Python programs

Setup: Use SQLite (No Installation Required)

To make this accessible:

- Use SQLite Online or DB Fiddle
- Or, run it locally with Python's `sqlite3` (built-in)

Create Sample Database & Tables

Let's create a basic schema with `Customers`, `Orders`, and `Products`.

```
-- Create Customers table
CREATE TABLE Customers (
    CustomerID INTEGER PRIMARY KEY,
    Name TEXT NOT NULL,
    Email TEXT UNIQUE
);

-- Create Orders table
CREATE TABLE Orders (
    OrderID INTEGER PRIMARY KEY,
    OrderDate TEXT NOT NULL,
    CustomerID INTEGER,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);

-- Insert sample customers
INSERT INTO Customers (Name, Email) VALUES
('Alice Johnson', 'alice@example.com'),
('Bob Smith', 'bob@example.com'),
('Carol White', 'carol@example.com');

-- Insert sample orders
INSERT INTO Orders (OrderDate, CustomerID) VALUES
('2025-07-01', 1),
('2025-07-02', 1),
('2025-07-03', 2);
```

Practice Tasks

Learners can now write SQL queries based on the schema above.

Task 1: Retrieve Customer Names with Orders

```
SELECT Customers.Name, Orders.OrderDate
FROM Customers
INNER JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
```

This gets only those customers who have placed at least one order.

Task 2: Find Customers Without Orders

```sql
SELECT Customers.Name
FROM Customers
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID
WHERE Orders.OrderID IS NULL;
```

This returns customers who don't have any orders. Useful for identifying inactive users.

Task 3: Count Orders Per Customer

```sql
SELECT Customers.Name, COUNT(Orders.OrderID) AS OrderCount
FROM Customers
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID
GROUP BY Customers.CustomerID;
```

Combines join with aggregation — helps find total orders per customer.

Python Integration: Accessing SQL from Python

Once a database is created (e.g., `sample.db`), you can run SQL queries directly in Python.

Example Python Script

```python
import sqlite3

# Connect to the SQLite database (creates file if it doesn't exist)
conn = sqlite3.connect('sample.db')
cursor = conn.cursor()

# Create tables
cursor.executescript("""
CREATE TABLE IF NOT EXISTS Customers (
    CustomerID INTEGER PRIMARY KEY,
    Name TEXT,
    Email TEXT
);

CREATE TABLE IF NOT EXISTS Orders (
    OrderID INTEGER PRIMARY KEY,
    OrderDate TEXT,
    CustomerID INTEGER,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
""")

# Insert data (optional if already exists)
cursor.execute("INSERT OR IGNORE INTO Customers (CustomerID, Name, Email) VALUES (1, 'Alice Jo
cursor.execute("INSERT OR IGNORE INTO Customers (CustomerID, Name, Email) VALUES (2, 'Bob Smit
```

```
cursor.execute("INSERT OR IGNORE INTO Orders (OrderID, OrderDate, CustomerID) VALUES (101, '202

# Run a SELECT query
cursor.execute("""
SELECT Customers.Name, Orders.OrderDate
FROM Customers
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID
""")

# Fetch and display results
results = cursor.fetchall()
for row in results:
    print(row)

conn.close()
```

This script creates a small local database, adds data, and runs a join query — great for practice and prototyping!

Bonus Assignments

1. Get the most recent order for each customer
2. Count how many customers placed multiple orders
3. Add a `Products` table and JOIN it to orders
4. Filter orders by date range (e.g., July 2025)
5. Export query results to CSV in Python

V.Homework

LESSON 14: INTRODUCTION TO SQL FOR PYTHON DEVELOPERS

Homework

Task: Write a series of SQL queries that:

1. Retrieve all customer names who have placed more than one order.
2. Insert a new order for a given customer.
3. Update an order's date.
4. Delete an order by ID.

Documentation Requirement: For each query, explain:

- What the query does
- Why it would be useful in a real application
- How it could be integrated with Python (e.g., `sqlite3`, `pandas.read_sql_query`)

Bonus: Export a SQL query result to a `.csv` using Python, e.g.:

```
import pandas as pd
df = pd.read_sql_query("SELECT * FROM Orders", conn)
df.to_csv('orders.csv', index=False)
```

Lesson 15

Ii. Advanced Sql Techniques

II. Advanced SQL Techniques

---

a. Complex Queries with Subqueries and Advanced Predicates

Setup: Tables & Data

```
-- USERS table
CREATE TABLE users (
    user_id SERIAL PRIMARY KEY,
    name VARCHAR(50),
    email VARCHAR(100)
);

-- ORDERS table
CREATE TABLE orders (
    order_id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(user_id),
    order_total DECIMAL(10, 2),
    order_date DATE
);

-- PRODUCTS table
CREATE TABLE products (
    product_id SERIAL PRIMARY KEY,
    product_name VARCHAR(100),
    price DECIMAL(10, 2),
    category VARCHAR(50)
);
-- Sample USERS
INSERT INTO users (name, email) VALUES
('Alice', 'alice@example.com'),
('Bob', 'bob@example.com'),
('Charlie', 'charlie@example.com');

-- Sample ORDERS
INSERT INTO orders (user_id, order_total, order_date) VALUES
```

```
(1, 120.50, '2023-11-01'),
(1, 75.00, '2023-11-10'),
(2, 50.00, '2023-11-05'),
(3, 200.00, '2023-11-15');

-- Sample PRODUCTS
INSERT INTO products (product_name, price, category) VALUES
('Keyboard', 25.00, 'Accessories'),
('Monitor', 150.00, 'Electronics'),
('Mouse', 20.00, 'Accessories'),
('Budget Monitor', 90.00, 'Discount'),
('High-End Monitor', 250.00, 'Electronics');
```

Example 1: Subquery to Find High-Spending Users

```
SELECT user_id, name
FROM users
WHERE user_id IN (
    SELECT user_id
    FROM orders
    GROUP BY user_id
    HAVING SUM(order_total) > (
        SELECT AVG(order_total) FROM orders
    )
);
```

What it does: Returns users whose total spending is above the average order value across all users.

Python Use Case: Analytics module in a web app to identify "VIP customers".

```
cursor.execute("""
    SELECT user_id, name
    FROM users
    WHERE user_id IN (
        SELECT user_id
        FROM orders
        GROUP BY user_id
        HAVING SUM(order_total) > (
            SELECT AVG(order_total) FROM orders
        )
    )
""")
vip_users = cursor.fetchall()
```

Example 2: Using `ALL` Predicate

```sql
SELECT product_name FROM products
WHERE price > ALL (
    SELECT price FROM products WHERE category = 'Discount'
);
```

What it does: Finds products priced higher than **all** products in the 'Discount' category.

Python Use Case: Recommendation system to promote premium products.

```python
cursor.execute("""
    SELECT product_name FROM products
    WHERE price > (
        SELECT price FROM products WHERE category = 'Discount'
    )
""")
premium_products = cursor.fetchall()
```

Example 3: `EXISTS` Predicate

```sql
SELECT name
FROM users u
WHERE EXISTS (
    SELECT 1 FROM orders o WHERE o.user_id = u.user_id AND o.order_total > 100
);
```

What it does: Returns users who have placed **at least one order** over $100.

Why `EXISTS` is efficient: Stops scanning as soon as one matching record is found.

Python Use Case: Conditional content delivery (e.g., "Thanks for your big purchase!").

Example 4: Correlated Subquery — Latest Order per User

```sql
SELECT u.name, o.order_id, o.order_date
FROM users u
JOIN orders o ON u.user_id = o.user_id
WHERE o.order_date = (
    SELECT MAX(order_date)
    FROM orders o2
    WHERE o2.user_id = u.user_id
);
```

What it does: For each user, returns only their **most recent order**.

Python Use Case: Generating personalized dashboards that show the most recent activity per user.

Example 5: Subquery in SELECT Clause — Total Spend per User

```
SELECT u.name,
       (SELECT SUM(order_total)
        FROM orders o
        WHERE o.user_id = u.user_id) AS total_spent
FROM users u;
```

What it does: Displays each user along with the **sum of their orders**, using a subquery in the `SELECT` clause.

Python Use Case: Used in email personalization ("Hi Alice, you've spent $XXX this month!")

Example 6: NOT EXISTS — Users With No Orders

```
SELECT name
FROM users u
WHERE NOT EXISTS (
    SELECT 1 FROM orders o WHERE o.user_id = u.user_id
);
```

What it does: Lists all users who have **never placed an order**.

Python Use Case: Useful for retention campaigns or reactivation emails.

Example 7: `IN` with Aggregate Filtering — Users with More Than One Order

```
SELECT name
FROM users
WHERE user_id IN (
    SELECT user_id
    FROM orders
    GROUP BY user_id
    HAVING COUNT(order_id) > 1
);
```

What it does: Returns users who have placed **more than one** order.

Python Use Case: Rewarding returning customers or tracking engagement.

Example 8: ALL with Correlated Comparison — Expensive Products Only

```
SELECT product_name
FROM products
WHERE price > (
    SELECT price FROM products WHERE category = 'Accessories'
);
```

What it does: Selects products more expensive than **every** item in the 'Accessories' category.

Python Use Case: Used for upselling strategies in e-commerce apps.

Example 9: Nested Subqueries — Top Spender

```
SELECT name
FROM users
WHERE user_id = (
    SELECT user_id
    FROM orders
    GROUP BY user_id
    ORDER BY SUM(order_total) DESC
    LIMIT 1
);
```

What it does: Returns the user who has spent the **most overall**.

Python Use Case: For leaderboard displays, loyalty programs, or admin analytics.

Example 10: EXISTS with JOIN — Users Who Paid Exactly What They Ordered

```
SELECT u.name
FROM users u
WHERE EXISTS (
    SELECT 1
    FROM orders o
    JOIN payments p ON o.order_id = p.order_id
    WHERE o.user_id = u.user_id AND o.order_total = p.amount
);
```

What it does: Finds users whose **orders were fully paid** (no discounts, no outstanding balances).

Python Use Case: Reconciliation logic or identifying complete transactions for reporting.

Example 11: Subquery in `FROM` Clause — Average Spend per User

```
SELECT name, avg_spent
FROM (
    SELECT u.user_id, u.name, AVG(o.order_total) AS avg_spent
    FROM users u
    JOIN orders o ON u.user_id = o.user_id
    GROUP BY u.user_id
) AS user_avg;
```

What it does: Calculates and displays average order value **per user**.

Python Use Case: Behavioral segmentation based on spending patterns.

b. Enhanced SQL Joins for Multi-Table Queries

Add PAYMENTS Table & Data

```
-- PAYMENTS table
CREATE TABLE payments (
    payment_id SERIAL PRIMARY KEY,
    order_id INTEGER REFERENCES orders(order_id),
    payment_date DATE,
    amount DECIMAL(10, 2)
);

-- Sample PAYMENTS
INSERT INTO payments (order_id, payment_date, amount) VALUES
(1, '2023-11-02', 120.50),
(2, '2023-11-11', 75.00),
(3, '2023-11-06', 50.00),
(4, '2023-11-16', 200.00);
```

Example 4: Join for Full Transaction History

```
SELECT u.name, o.order_id, o.order_total, p.payment_date, p.amount
FROM users u
JOIN orders o ON u.user_id = o.user_id
JOIN payments p ON o.order_id = p.order_id;
```

What it does: Combines user names, their orders, and payment details in one view.

Python Use Case: Data export tool or dashboard backend showing detailed purchase logs.

```
cursor.execute("""
    SELECT u.name, o.order_id, o.order_total, p.payment_date, p.amount
    FROM users u
    JOIN orders o ON u.user_id = o.user_id
    JOIN payments p ON o.order_id = p.order_id
""")
transactions = cursor.fetchall()
```

Example 5: LEFT JOIN to Show Users With or Without Orders

```
SELECT u.name, o.order_id
FROM users u
LEFT JOIN orders o ON u.user_id = o.user_id;
```

What it does: Returns all users, with their orders if any — includes users who haven't ordered yet.

Python Use Case: Generating reports for user engagement (e.g., for marketing or email campaigns).

```
cursor.execute("""
    SELECT u.name, o.order_id, o.order_total
    FROM users u
    LEFT JOIN orders o ON u.user_id = o.user_id
""")
results = cursor.fetchall()
for row in results:
    print(row)
```

Example 6: Aggregated Joins

```
SELECT u.name, COUNT(o.order_id) AS total_orders, SUM(o.order_total) AS total_spent
FROM users u
LEFT JOIN orders o ON u.user_id = o.user_id
GROUP BY u.user_id;
```

What it does: Shows number of orders and total spend per user.

Python Use Case: Dashboard summary for business stakeholders or automated email personalization.

Example 7: LEFT JOIN – Users with Their Orders (Even if None)

```
SELECT u.name, o.order_id, o.order_total
FROM users u
LEFT JOIN orders o ON u.user_id = o.user_id;
```

What it does: Shows all users, even if they **haven't placed any orders** (`order_id` will be NULL in that case).

Use Case: Display all users in an admin report, highlighting inactive ones.

Example 8: RIGHT JOIN – Orders and Associated Users (Even if User is Missing)

> **Note:** Not all databases (e.g., SQLite) support `RIGHT JOIN`. If unsupported, flip tables and use `LEFT JOIN`.

```
SELECT o.order_id, o.order_total, u.name
FROM orders o
RIGHT JOIN users u ON o.user_id = u.user_id;
```

What it does: Shows all users, with their orders. Effectively same as `LEFT JOIN users → orders`.

Use Case: In a report, ensure all users are listed regardless of whether they've made purchases.

Example 9: FULL OUTER JOIN – All Users and All Orders, Matched if Possible

```
SELECT u.name, o.order_id, o.order_total
FROM users u
FULL OUTER JOIN orders o ON u.user_id = o.user_id;
```

What it does: Combines all data from both `users` and `orders`. If there's no match, the unmatched side gets NULL.

Use Case: Reconciling data between systems (e.g., CRM vs sales database).

Example 10: CROSS JOIN – All Combinations of Users and Products

```
SELECT u.name, p.product_name
FROM users u
CROSS JOIN products p;
```

What it does: Generates a combination of every user with every product.

Use Case: Simulate "what-if" scenarios (e.g., testing personalized recommendations).

Warning: Can grow large fast. `3 users × 5 products = 15 rows`.

Example 11: JOIN with Filter – Orders Over $100 With User Info

```
SELECT u.name, o.order_id, o.order_total
FROM users u
JOIN orders o ON u.user_id = o.user_id
WHERE o.order_total > 100;
```

What it does: Returns only users who placed **orders over $100**, with details.

Use Case: Loyalty programs, bonus triggers, or fraud detection.

Example 12: Multiple Table JOIN – Full Payment Info with User

```
SELECT u.name, o.order_id, o.order_total, p.amount, p.payment_date
FROM users u
JOIN orders o ON u.user_id = o.user_id
JOIN payments p ON o.order_id = p.order_id;
```

What it does: Shows complete transaction records: who placed the order, how much, and when they paid.

Use Case: Generating detailed invoices or admin financial reports.

Example 13: JOIN with Aggregation – Total Payments Per User

```
SELECT u.name, SUM(p.amount) AS total_paid
FROM users u
JOIN orders o ON u.user_id = o.user_id
JOIN payments p ON o.order_id = p.order_id
GROUP BY u.name;
```

What it does: Aggregates total payments made **per user**.

Use Case: Customer lifetime value tracking or financial summary views.

Summary: JOIN Selection Guide

Join Type	Description
`INNER JOIN`	Returns only matching rows in both tables
`LEFT JOIN`	All rows from the **left** table + matching from right (if any)
`RIGHT JOIN`	All rows from the **right** table + matching from left (if any)
`FULL OUTER JOIN`	All rows from both tables, matched when possible, `NULL` otherwise
`CROSS JOIN`	Cartesian product (every row of A paired with every row of B)

Iii. Query Optimization For Python Applications

III. Query Optimization for Python Applications

Optimizing SQL queries isn't just about database performance—it's about **writing Python applications that are faster, safer, and more scalable**. This section explains **how indexes, query limits, prepared statements, and connection pooling** contribute to efficient Python+SQL workflows.

---

a. Indexing and Performance

What is an Index? An **index** is like a lookup table used by the database to **quickly locate rows** without scanning the entire table. Think of it as a book's table of contents.

- Without an index: the DB must **scan every row** (sequential scan).
- With an index: it **jumps directly** to matching entries (index scan).

Creating an Index Example: Create an index on `user_id` in the `orders` table.

```
CREATE INDEX idx_user_id ON orders(user_id);
```

Why?

If you're frequently filtering orders by user in Python:

```
cursor.execute("SELECT * FROM orders WHERE user_id = %s", (some_user_id,))
```

…then indexing `user_id` **dramatically speeds up** the query by avoiding full table scans.

Measuring with EXPLAIN Use `EXPLAIN` before your query to see how it's executed:

```
EXPLAIN SELECT * FROM orders WHERE user_id = 1;
```

Output:

```
Index Scan using idx_user_id on orders  (cost=0.15..8.27 rows=1 width=32)
```

Means it's **using the index** – much faster than a "Seq Scan".

Python Example with Indexed Query

```
cursor.execute("EXPLAIN SELECT * FROM orders WHERE user_id = %s", (1,))
for row in cursor.fetchall():
    print(row[0])
```

Best Practice: Index columns used in:

- WHERE conditions
- JOIN keys
- ORDER BY clauses

Avoid indexing:

- Small tables
- Highly volatile columns (frequent updates)
- Columns with low selectivity (e.g. `is_active` with 99% TRUE)

---

a2. Other Types of indexes

Single-Column Index

- Index on one column (e.g., `user_id`)
- Best for queries like:

  ```
  SELECT * FROM orders WHERE user_id = 3;
  ```

```
CREATE INDEX idx_user_id ON orders(user_id);
```

Multi-Column (Composite) Index

- Index on **two or more columns** used together in WHERE or JOIN clauses.
- Example: Speed up filtering on `user_id` **and** `order_date`.

```
CREATE INDEX idx_user_date ON orders(user_id, order_date);
```

Useful for queries like:

```
SELECT * FROM orders
WHERE user_id = 2 AND order_date > '2024-01-01';
```

Order matters in composite indexes.

Unique Index

- Ensures values in a column are unique (like emails).
- Often added automatically with `PRIMARY KEY` or `UNIQUE`.

```
CREATE UNIQUE INDEX idx_unique_email ON users(email);
```

Prevents duplicate users based on email.

Partial Index

- Only indexes **part** of a table (rows that meet a condition).
- Useful for optimizing **frequently queried subsets**.

```
CREATE INDEX idx_high_value_orders
ON orders(order_total)
WHERE order_total > 100;
```

Speeds up:

```
SELECT * FROM orders WHERE order_total > 100;
```

Covering Index

- Index that includes **all columns** needed by a query.
- Avoids going back to the main table (index-only scan).

```
CREATE INDEX idx_cover_user_orders ON orders(user_id, order_total);
```

Covers:

```
SELECT user_id, order_total FROM orders WHERE user_id = 2;
```

Choosing the Right Index

Query Type	Index Recommendation
Filter by one column	Single-column index
Filter by two+ columns	Composite index
Frequently queried value range	Partial index
Columns always returned together	Covering index
Enforce uniqueness (e.g. email)	Unique index

Using EXPLAIN to See Index Use

```
EXPLAIN SELECT * FROM orders WHERE user_id = 2;
```

Look for: `Index Scan using idx_user_id` Avoid: `Seq Scan` unless table is very small

Reminder: Indexes Improve Reads, But...

- **Slow down INSERTs/UPDATEs**
- **Consume disk space**
- **Need maintenance (reindexing occasionally)**

---

b. Limiting Result Sets

When fetching from large tables (e.g., logs, transactions), always **limit rows** to avoid memory issues.

```
SELECT * FROM orders LIMIT 100 OFFSET 0;
```

Python Tip: Use Pagination or Generators

```
def get_orders_paginated(offset=0, limit=100):
    cursor.execute("SELECT * FROM orders LIMIT %s OFFSET %s", (limit, offset))
    return cursor.fetchall()
```

Also consider:

- Using cursors (`fetchmany()`)
- Streaming data if using `psycopg2` with `named cursors`

---

c. Prepared Statements and Parameterization

Why Use Parameters?

- Prevent **SQL injection**
- Improve **query plan reuse** (database caches execution plan)
- Clean separation of SQL logic and values

Unsafe Example (Don't do this!)

```
email = "bob@example.com"
cursor.execute(f"SELECT * FROM users WHERE email = '{email}'")  # BAD
```

If `email` is `"bob@example.com' OR '1'='1"` – you've just been hacked.

Safe Parameterized Version

```
cursor.execute("SELECT * FROM users WHERE email = %s", (email,))
user = cursor.fetchone()
```

Works similarly with `INSERT`, `UPDATE`, etc.:

```
cursor.execute(
    "INSERT INTO payments (order_id, payment_date, amount) VALUES (%s, %s, %s)",
    (2, '2023-11-12', 75.00)
)
```

---

d. Connection Pooling

Creating new DB connections is expensive. **Pooling** allows reuse of existing connections.

Example with `psycopg2`:

```
from psycopg2 import pool

connection_pool = pool.SimpleConnectionPool(
    1, 10,
    user="postgres",
    password="pass",
    host="localhost",
    database="mydb"
)

# Reuse a connection
conn = connection_pool.getconn()
cursor = conn.cursor()
cursor.execute("SELECT * FROM users")
```

```
...
connection_pool.putconn(conn)  # Return to pool
```

Why it matters:

- Web apps (e.g., Flask, FastAPI) handle many requests in parallel.
- Pooling ensures connections are reused, not re-opened each time.

Summary Table: Optimization Techniques

Technique	Benefit	Python Example
Indexing	Speeds up filtering/searches	`WHERE user_id = %s`
EXPLAIN/ANALYZE	Reveals performance bottlenecks	`cursor.execute("EXPLAIN ...")`
LIMIT/OFFSET	Prevents memory overload	Pagination with `LIMIT %s OFFSET %s`
Parameterized Queries	Safer and faster	`cursor.execute(..., (value,))`
Connection Pooling	Efficient resource usage in servers	`SimpleConnectionPool`

Exercise

IV. Hands-on Exercise: Advanced SQL Techniques & Index Optimization

---

**** Goal:****

Apply **advanced SQL features**—including **subqueries**, **correlated subqueries**, **joins**, **advanced predicates**, and **index creation**—to solve business-relevant problems using the shared sample dataset.

**** Dataset Summary (Reminder)****

You're working with the following tables:

- `users(user_id, name, email)`
- `orders(order_id, user_id, order_total, order_date)`
- `payments(payment_id, order_id, payment_date, amount)`
- `products(product_id, product_name, price, category)`

Sample data is described as SQL in earlier part of this lesson.

**** Tasks & Queries****

Task 1: Identify High-Spending Users Using a Subquery Query:

```
SELECT user_id, name
FROM users
WHERE user_id IN (
    SELECT user_id
    FROM orders
    GROUP BY user_id
    HAVING SUM(order_total) > (
        SELECT AVG(total_sum)
        FROM (
            SELECT user_id, SUM(order_total) AS total_sum
            FROM orders
            GROUP BY user_id
        ) AS user_totals
    )
);
```

Explanation: Finds users whose total spending exceeds the **average total spending per user**.

Task 2: Correlated Subquery – Most Recent Order per User Query:

```
SELECT u.name, o.order_id, o.order_date
FROM users u
JOIN orders o ON u.user_id = o.user_id
WHERE o.order_date = (
    SELECT MAX(o2.order_date)
    FROM orders o2
    WHERE o2.user_id = u.user_id
);
```

Explanation: Returns each user's **latest order** using a correlated subquery.

Task 3: Create an Index to Optimize Task 2 Query:

```
CREATE INDEX idx_user_order_date ON orders(user_id, order_date);
```

Explanation: This **composite index** speeds up filtering and sorting by `user_id` and `order_date`, which Task 2 depends on.

Task 4: Use `ALL` Predicate – Premium Products Query:

```
SELECT product_name, price
FROM products
WHERE price > ALL (
```

```
    SELECT price FROM products WHERE category = 'Discount'
);
```

Explanation: Returns all products priced higher than **every product** in the 'Discount' category.

Task 5: Use `EXISTS` – Users with Orders Over $100 Query:

```
SELECT name
FROM users u
WHERE EXISTS (
    SELECT 1 FROM orders o
    WHERE o.user_id = u.user_id AND o.order_total > 100
);
```

Explanation: Returns users who have made **at least one high-value purchase** over $100.

Task 6: Create a Partial Index to Optimize Task 5 (This part is optional) Query:

```
CREATE INDEX idx_high_order_total
ON orders(order_total)
WHERE order_total > 100;
```

Explanation: This **partial index** is efficient for queries that filter orders **only when `order_total > 100`**.

Task 7: Rewrite a Subquery Using JOIN – Users Who Made Payments Original Query (subquery style):

```
SELECT name
FROM users
WHERE user_id IN (
    SELECT o.user_id
    FROM orders o
    JOIN payments p ON o.order_id = p.order_id
);
```

Rewritten with JOIN:

```
SELECT DISTINCT u.name
FROM users u
JOIN orders o ON u.user_id = o.user_id
JOIN payments p ON o.order_id = p.order_id;
```

Explanation: Retrieves the same result using **explicit joins**; `DISTINCT` avoids duplicate names for users with multiple payments.

Outline

LESSON 15: ADVANCED SQL CONCEPTS WITH PYTHON APPLICATIONS

1. Recap of Previous Lesson

- Review of intermediate SQL concepts and basic Python-SQL integration.
- Overview of database drivers (e.g., `sqlite3`, `psycopg2`, `SQLAlchemy`).

---

2. Advanced SQL Techniques

a. Complex Queries with Subqueries and Advanced Predicates

- **Subqueries**:
  - Example: Fetch users who placed orders greater than the average order value.

```
SELECT user_id, name
FROM users
WHERE user_id IN (
    SELECT user_id
    FROM orders
    GROUP BY user_id
    HAVING SUM(order_total) > (
        SELECT AVG(order_total) FROM orders
    )
);
```

Python Context: Use subqueries within Python functions that require filtering based on aggregate data.

- **Advanced Predicates** (`IN`, `EXISTS`, `ANY`, `ALL`):
  - Example:

```
SELECT product_name FROM products
WHERE price > ALL (
    SELECT price FROM products WHERE category = 'Discount'
);
```

Explanation: Useful for comparative filtering in analytics modules.

---

b. Enhanced SQL Joins for Multi-Table Queries

- **Types of Joins**: `INNER`, `LEFT`, `RIGHT`, `FULL OUTER`, `CROSS`
- **Use Case in Python**:
  - Joining `users`, `orders`, and `payments` for a full transaction history:

```sql
SELECT u.name, o.order_id, p.payment_date, p.amount
FROM users u
JOIN orders o ON u.user_id = o.user_id
JOIN payments p ON o.order_id = p.order_id;
```

Python Example:

```python
cursor.execute("""SELECT u.name, o.order_id, p.payment_date, p.amount
                  FROM users u
                  JOIN orders o ON u.user_id = o.user_id
                  JOIN payments p ON o.order_id = p.order_id""")
results = cursor.fetchall()
```

---

3. Query Optimization for Python Applications

a. Indexing and Performance

- Use `EXPLAIN` or `EXPLAIN ANALYZE` to understand query execution plans.
- Add indexes on frequently queried columns.

  ```sql
  CREATE INDEX idx_user_id ON orders(user_id);
  ```

  Python Benefit: Reduces API latency in data-heavy endpoints.

b. Limiting Result Sets

- Avoid loading large datasets into memory:

  ```sql
  SELECT * FROM logs LIMIT 100 OFFSET 0;
  ```

  Python Tip: Use generators or pagination when working with large queries.

c. Prepared Statements and Parameterization

- Prevents SQL injection and speeds up repeated queries.

  ```python
  cursor.execute("SELECT * FROM users WHERE email = %s", (user_email,))
  ```

d. Connection Pooling (e.g., with SQLAlchemy or `psycopg2.pool`)

- Reuses DB connections across requests in web apps.

  ```python
  from psycopg2 import pool
  connection_pool = pool.SimpleConnectionPool(1, 10, user="postgres", ...)
  ```

---

4. Hands-on Exercise

Homework

Assignment:

Lesson 16

Ii. Introduction And Database Setup

LESSON 16: INTEGRATING SQL WITH PYTHON

II. Introduction and Database Setup

Topics Covered:

- Recap of previous lesson (data handling, files, or structures)
- Introduction to relational databases and SQL
- Using Python to create and connect to SQLite databases

Key Concept

A **relational database** is a structured way to store data in tables (rows & columns) with relationships between them. **SQL (Structured Query Language)** is the standard way to interact with databases: insert, retrieve, update, and delete data. Python's **`sqlite3` module** allows you to handle a relational database **without leaving Python**, perfect for small to medium apps.

Tool of Choice: `sqlite3`

- Built into Python (no installation required)
- Lightweight, file-based
- SQL standard support
- Suitable for demos, small apps, and prototyping

---

1. Creating a SQLite Database and Table

How SQLite Creates a Database

```
import sqlite3

# This will create 'lesson16.db' in the same folder if it doesn't exist
conn = sqlite3.connect('lesson16.db')
print("Database 'lesson16.db' created or opened successfully.")
conn.close()
```

> **If the file `lesson16.db` exists → it connects to it. If it doesn't exist → SQLite creates it.**

How to create a Table if Not Exists

```
def initialize_database():
    conn = sqlite3.connect('lesson16.db')
    cursor = conn.cursor()

    # Create 'users' table if it doesn't exist already
    cursor.execute('''
    CREATE TABLE IF NOT EXISTS users (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT NOT NULL,
        age INTEGER NOT NULL
    )
    ''')

    conn.commit()
    conn.close()
    print("Database initialized and 'users' table ensured.")

# Initialize the database and table
initialize_database()
```

Together in 4 steps with connection open and close

```
import sqlite3

#  Connect to (or create) a database file
conn = sqlite3.connect('lesson16.db')

cursor = conn.cursor()

cursor.execute('''
CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    age INTEGER NOT NULL
)
''')

# Commit changes and close connection
conn.commit()
conn.close()

print("Database and 'users' table created successfully.")
```

What this code does:

- **Connects** to `lesson16.db` (creates if not existing)

- **Creates** a `users` table with:
  - `id` as an auto-incrementing primary key
  - `name` as text (non-null)
  - `age` as integer (non-null)
- **Commits** changes
- **Closes** the connection

---

2. Repeated Connections — Safe Way with Context Manager

```
def create_user(name, age):
    with sqlite3.connect('lesson16.db') as conn:
        cursor = conn.cursor()
        cursor.execute('INSERT INTO users (name, age) VALUES (?, ?)', (name, age))
        conn.commit()
    print(f"User '{name}' added successfully.")

# Example Usage
create_user("Alice", 30)
create_user("Bob", 25)
```

Why use `with` block?

- Automatically closes the connection even if an error occurs.
- Safer and cleaner than manual close.

---

3. Reading Data from the Table

```
def fetch_all_users():
    with sqlite3.connect('lesson16.db') as conn:
        cursor = conn.cursor()
        cursor.execute('SELECT * FROM users')
        users = cursor.fetchall()
    return users

print(fetch_all_users())
```

Summary of Basic Database Operations

Operation	Python Example
Create Table	`cursor.execute('CREATE TABLE...')`
Insert Data	`cursor.execute('INSERT INTO...')`
Select Data	`cursor.execute('SELECT * FROM...')`

Operation	Python Example
Commit Changes	`conn.commit()`
Close Connection	`conn.close()` or use `with` block

Important: Simple Error Handling Example

```
def insert_user_safe(name, age):
    try:
        with sqlite3.connect('lesson16.db') as conn:
            cursor = conn.cursor()
            cursor.execute('INSERT INTO users (name, age) VALUES (?, ?)', (name, age))
            conn.commit()
        print(f"Inserted user: {name}")
    except sqlite3.Error as e:
        print(f"An error occurred: {e}")

insert_user_safe("Charlie", 40)
```

Recap:

- You can **fully control a SQLite database directly from Python**.
- **Connection → Cursor → Execute SQL → Commit → Close** is the typical workflow.
- Using functions makes database operations reusable and clean.
- Using **parameterized queries** (`?` placeholders) protects against SQL injection.
- Context managers (`with` block) ensure safe connection handling.

Iii. Performing Sql Operations In Python

III. Performing SQL Operations in Python

Topics Covered:

- SQL Statements: `SELECT`, `INSERT`, `UPDATE`, `DELETE`
- SQL Clauses: `WHERE`, `ORDER BY`, `LIMIT`
- Predicates for filtering data
- How to safely pass data to SQL queries (prevent SQL injection)

Key Concept

You can perform all major SQL operations directly by sending SQL statements as strings from Python using **`cursor.execute()`**.

- SQL operations are dynamic — parameters are passed at runtime.
- Always use placeholders (`?`) in SQLite queries to prevent SQL injection.

Basic SQL Operations in Python

INSERT Operation (Adding Data)

```
def add_user(name, age):
    with sqlite3.connect('lesson16.db') as conn:
        cursor = conn.cursor()
        cursor.execute('INSERT INTO users (name, age) VALUES (?, ?)', (name, age))
        conn.commit()
    print(f"User '{name}' added successfully.")

# Example
add_user("David", 27)
add_user("Eve", 34)
```

SELECT Operation (Reading Data)

```
def get_all_users():
    with sqlite3.connect('lesson16.db') as conn:
        cursor = conn.cursor()
        cursor.execute('SELECT * FROM users')
        users = cursor.fetchall()
    return users

# Example
print("All Users:", get_all_users())
```

SELECT with WHERE Clause (Filtering Data)

```
def get_users_above_age(min_age):
    with sqlite3.connect('lesson16.db') as conn:
        cursor = conn.cursor()
        cursor.execute('SELECT * FROM users WHERE age > ?', (min_age,))
        users = cursor.fetchall()
    return users

# Example
print("Users older than 30:", get_users_above_age(30))
```

SELECT with ORDER BY and LIMIT Clauses

```
def get_youngest_users(limit):
    with sqlite3.connect('lesson16.db') as conn:
        cursor = conn.cursor()
        cursor.execute('SELECT * FROM users ORDER BY age ASC LIMIT ?', (limit,))
        users = cursor.fetchall()
    return users

# Example
print("Top 2 youngest users:", get_youngest_users(2))
```

UPDATE Operation (Modifying Data)

```
def update_user_age(name, new_age):
    with sqlite3.connect('lesson16.db') as conn:
        cursor = conn.cursor()
        cursor.execute('UPDATE users SET age = ? WHERE name = ?', (new_age, name))
        conn.commit()
    print(f"Updated '{name}' to age {new_age}.")

# Example
update_user_age("David", 29)
print(get_all_users())
```

DELETE Operation (Removing Data)

```
def delete_user(name):
    with sqlite3.connect('lesson16.db') as conn:
        cursor = conn.cursor()
        cursor.execute('DELETE FROM users WHERE name = ?', (name,))
        conn.commit()
    print(f"User '{name}' deleted.")

# Example
delete_user("Eve")
print(get_all_users())
```

Summary of SQL Operations with Examples

Operation	SQL Syntax	Python Example
INSERT	`INSERT INTO table (cols) VALUES (?, ?)`	`cursor.execute('INSERT INTO...', (value1, value2))`
SELECT	`SELECT * FROM table`	`cursor.execute('SELECT * FROM...')`

Operation	SQL Syntax	Python Example
UPDATE	`UPDATE table SET col = ? WHERE condition`	`cursor.execute('UPDATE table SET...')`
DELETE	`DELETE FROM table WHERE condition`	`cursor.execute('DELETE FROM table WHERE...')`

Why Use Placeholders (?)

- Protects against **SQL injection**
- Keeps queries readable and maintainable
- Avoids syntax errors with dynamic data

Practical Example — Combined Operations

```
def user_workflow_demo():
    add_user("Frank", 40)
    print("All users after adding Frank:", get_all_users())

    update_user_age("Frank", 41)
    print("All users after updating Frank:", get_all_users())

    delete_user("Frank")
    print("All users after deleting Frank:", get_all_users())

# Run the demo
user_workflow_demo()
```

Recap:

- You can fully control database data — insert, retrieve, update, delete — using SQL commands embedded in Python.
- Using **functions** for each operation keeps code modular and reusable.
- SQL clauses like `WHERE`, `ORDER BY`, and `LIMIT` make data querying powerful and flexible.
- Always commit your changes after modifying data.

Iv. Advanced Queries And Integration With Functions Pandas

IV. Advanced Queries and Integration with Functions & Pandas

Topics Covered:

- Using SQL **JOINS** to link data across tables
- Embedding SQL logic in **reusable Python functions**
- Reading SQL results into **Pandas DataFrames**
- Using Pandas for simple data exploration

Key Concept

A relational database often contains **multiple related tables**. **JOIN operations** allow you to retrieve combined data based on relationships (e.g., users with their orders). Pandas is a powerful library that can turn query results into DataFrames for further analysis.

---

1. Setting Up Related Tables

Users Table — (already created in earlier sections)

Creating a Related `orders` Table

```
def create_orders_table():
    with sqlite3.connect('lesson16.db') as conn:
        cursor = conn.cursor()
        cursor.execute('''
        CREATE TABLE IF NOT EXISTS orders (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            user_id INTEGER NOT NULL,
            product TEXT NOT NULL,
            FOREIGN KEY(user_id) REFERENCES users(id)
        )
        ''')
        conn.commit()
    print("Orders table created or confirmed existing.")

# Run once to ensure table exists
create_orders_table()
```

---

2. Insert Sample Data for JOIN Demonstration

```
def add_order(user_id, product):
    with sqlite3.connect('lesson16.db') as conn:
        cursor = conn.cursor()
        cursor.execute('INSERT INTO orders (user_id, product) VALUES (?, ?)', (user_id, product
        conn.commit()
    print(f"Order '{product}' added for user ID {user_id}.")

# Example Usage
```

```
add_user("Charlie", 32)
add_user("Dana", 29)

add_order(3, "Smartphone")  # Assuming Charlie is ID 3
add_order(4, "Tablet")      # Assuming Dana is ID 4
```

---

3. Performing INNER JOIN with Python & SQLite

Inner Join Function to Fetch Combined Data

```
def fetch_users_with_orders():
    with sqlite3.connect('lesson16.db') as conn:
        cursor = conn.cursor()
        cursor.execute('''
        SELECT users.name, users.age, orders.product
        FROM users
        INNER JOIN orders ON users.id = orders.user_id
        ''')
        results = cursor.fetchall()
    return results

# Example Output
print("Users with their Orders:")
for row in fetch_users_with_orders():
    print(row)
```

What Does INNER JOIN Do?

- Combines **users** and **orders** based on `user_id`
- Only returns rows where the user has at least one order
- Allows pulling relational data in a single query

---

4. Integrating with Pandas for Data Analysis

```
import pandas as pd

def get_users_orders_dataframe():
    with sqlite3.connect('lesson16.db') as conn:
        df = pd.read_sql_query('''
        SELECT users.id, users.name, users.age, orders.product
        FROM users
        INNER JOIN orders ON users.id = orders.user_id
        ''', conn)
    return df

# Example
```

```
df_users_orders = get_users_orders_dataframe()
print("\nUsers and their Orders DataFrame:")
print(df_users_orders)
```

---

5. (optional): Use Pandas to Explore Data

```
# Count orders per user
order_counts = df_users_orders.groupby('name').size().reset_index(name='Order Count')
print("\nOrder Count per User:")
print(order_counts)

# Filter users older than 30 with their orders
older_users_orders = df_users_orders[df_users_orders['age'] > 30]
print("\nUsers older than 30 with their Orders:")
print(older_users_orders)
```

Concepts

Concept	How It's Done
Create Related Tables	Define tables with `FOREIGN KEY` constraints
Insert Related Data	Insert into both parent and child tables
INNER JOIN	Combine rows across tables with matching keys
Embed SQL in Functions	Wrap SQL logic inside Python functions
Use Pandas	Import query results as DataFrames for analysis

Recap

- SQL JOINs allow you to pull related data across multiple tables.
- Embedding SQL queries in Python functions makes your code modular and reusable.
- Using Pandas with SQL result sets turns your data into powerful, easy-to-analyze tables.
- Pandas supports filtering, grouping, and summarizing SQL data within Python.
-

Sqlalchemy_Additional

SQLAlchemy

The Python SQL Toolkit and Object Relational Mapper SQLAlchemy is the Python SQL toolkit and Object Relational Mapper that gives application developers the full power and flexibility of SQL.

It provides a full suite of well known enterprise-level persistence patterns, designed for efficient and high-performing database access, adapted into a simple and Pythonic domain language. (source: www.sqlalchemy.org)

Install SQLAlchemy

```
pip install sqlalchemy
```

Basic SQLAlchemy Concepts

Concept	Purpose
Engine	Core interface to the database (manages connections)
Session	Manages ORM transactions (unit of work)
Base	Declarative base class for ORM models
Model	Python class mapped to a table

SQLite Connection String Example

```python
from sqlalchemy import create_engine

# SQLite in-memory
engine = create_engine('sqlite:///:memory:')

# SQLite file database
engine = create_engine('sqlite:///my_database.db')
```

Declarative ORM Model Example

```python
from sqlalchemy.orm import declarative_base, sessionmaker
from sqlalchemy import Column, Integer, String

Base = declarative_base()

class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(String)
    email = Column(String)
```

Create Tables and Session

```
# Create tables in the database
Base.metadata.create_all(engine)

# Create a session
Session = sessionmaker(bind=engine)
session = Session()
```

CRUD Operations Example

```
# Create
new_user = User(name='Alice', email='alice@example.com')
session.add(new_user)
session.commit()

# Read
users = session.query(User).all()

# Update
user = session.query(User).filter_by(name='Alice').first()
user.email = 'alice@newdomain.com'
session.commit()

# Delete
session.delete(user)
session.commit()
```

Summary

- Use `create_engine()` with SQLite connection string.
- Define models by inheriting from `Base`.
- Use `Session` to interact with the DB.
- Call `Base.metadata.create_all(engine)` to create tables.
- Perform CRUD operations with `session`.

Sqlalchemy_Short

SQLAlchemy

The Python SQL Toolkit and Object Relational Mapper SQLAlchemy is the Python SQL toolkit and Object Relational Mapper that gives application developers the full power and flexibility of SQL.

It provides a full suite of well known enterprise-level persistence patterns, designed for efficient and high-performing database access, adapted into a simple and Pythonic domain language. (source: www.sqlalchemy.org)

Install SQLAlchemy

```
pip install sqlalchemy
```

Basic SQLAlchemy Concepts

Concept	Purpose
Engine	Core interface to the database (manages connections)
Session	Manages ORM transactions (unit of work)
Base	Declarative base class for ORM models
Model	Python class mapped to a table

ORM like / without typical SQL

```python
from sqlalchemy import create_engine
from sqlalchemy.orm import declarative_base, sessionmaker
from sqlalchemy import Column, Integer, String
import pandas as pd

engine = create_engine('sqlite:///lesson16.db')

Base = declarative_base()

class User(Base):
    __tablename__ = 'users3'

    id = Column(Integer, primary_key=True)
    name = Column(String)
    email = Column(String)


Base.metadata.create_all(engine)

# Create a session
Session = sessionmaker(bind=engine)
session = Session()

# Define and add new user
new_user = User(name='Alice', email='alice@example.com')
session.add(new_user)

# Get all users
users_results = session.query(User).all()
session.commit()
```

```
# Convert results to Panzdas
data = [{'id': u.id, 'name': u.name, 'email': u.email} for u in users_results]


df = pd.DataFrame(data)
print(df)
```

with just typical SQL

```
from sqlalchemy import create_engine, text
import pandas as pd

engine = create_engine('sqlite:///lesson16.db')

with engine.connect() as conn:
    result = conn.execute(text("SELECT id, name, email FROM users3"))
    df = pd.DataFrame(result.mappings().all())
    print(df)
```

Summary

- Use `create_engine()` with SQLite connection string.
- Define models by inheriting from `Base`.
- Use `Session` to interact with the DB.
- Call `Base.metadata.create_all(engine)` to create tables.
- Perform CRUD operations with `session`.

V. Hands-On Practice And Homework Assignment

V. Hands-On Practice and Homework Assignment

Purpose of this Section

- Practice transactional SQL operations directly in Python
- Capture and verify affected rows after changes
- Encourage building a real-life mini-application with SQL and Python
- Use **Pandas** for data presentation and filtering

Key Concept: Transactions & Fetching Results

- A **transaction** groups one or more SQL operations together.
- You `COMMIT` to apply changes permanently.
- After an update, you often want to **fetch the affected data** for validation or further processing.

- This pattern is critical for building reliable applications.

Hands-On Demo: Transactional Update and Fetch

Function Example — Update and Fetch

```
def update_user_age_and_get(user_id, new_age):
    with sqlite3.connect('lesson16.db') as conn:
        cursor = conn.cursor()

        # Step 1: Perform UPDATE
        cursor.execute('UPDATE users SET age = ? WHERE id = ?', (new_age, user_id))
        conn.commit()

        # Step 2: Fetch the updated user
        cursor.execute('SELECT * FROM users WHERE id = ?', (user_id,))
        user = cursor.fetchone()

    return user

# Example Use
result = update_user_age_and_get(1, 36)
print("Updated User:", result)
```

Function Example — Delete and Confirm Deletion

```
def delete_user_and_confirm(user_id):
    with sqlite3.connect('lesson16.db') as conn:
        cursor = conn.cursor()

        # Step 1: Delete the user
        cursor.execute('DELETE FROM users WHERE id = ?', (user_id,))
        conn.commit()

        # Step 2: Check if user still exists
        cursor.execute('SELECT * FROM users WHERE id = ?', (user_id,))
        user = cursor.fetchone()

    return "Deleted Successfully" if user is None else "Deletion Failed"

# Example Use
print(delete_user_and_confirm(4))
```

Function Example — Insert New User with Transaction Confirmation

```
def insert_user_and_return_id(name, age):
    with sqlite3.connect('lesson16.db') as conn:
        cursor = conn.cursor()
        cursor.execute('INSERT INTO users (name, age) VALUES (?, ?)', (name, age))
        conn.commit()
        return cursor.lastrowid

# Example Use
new_id = insert_user_and_return_id("Grace", 31)
print(f"New user inserted with ID: {new_id}")
```

Bonus Example — Fetch Updated Data into Pandas DataFrame

```
import pandas as pd

def get_all_users_dataframe():
    with sqlite3.connect('lesson16.db') as conn:
        df = pd.read_sql_query('SELECT * FROM users', conn)
    return df

# Example Use
print(get_all_users_dataframe())
```

Homework Assignment: Build a Complete Python Mini-App

Objective:

Build a **Python application** that:

- Connects to an SQLite database
- Manages **customers** and **orders** using SQL operations
- Presents data with **Pandas**

Required Features

Feature	Description
Database Connection	Use `sqlite3` to connect to a local DB
Table Creation	Create at least two related tables (e.g., `customers` and `orders`)
SQL Operations	Perform `INSERT`, `SELECT`, `UPDATE`, and `DELETE`
JOIN	Implement at least one `INNER JOIN` between tables
Transactional Update	Perform an `UPDATE` and fetch the affected rows
Pandas Integration	Fetch data with `pandas.read_sql_query()` and display

Added Challenges (Optional)

- Use `ORDER BY` or `GROUP BY` with Pandas
- Handle SQL errors with `try-except`
- Use functions for every major operation (CRUD pattern)

Example Homework Structure Suggestion

```
project_folder/
   app.py                # Your main Python app with functions
   requirements.txt      # Optional, list of libraries (e.g., pandas)
   README.md             # Explain project, logic, SQL usage
   lesson16.db           # Your SQLite database file (auto-created)
   sample_output.png     # Optional, sample Pandas output screenshot
```

Outline

LESSON 16: INTEGRATING SQL WITH PYTHON

Tools Required: Python 3.x, SQLite (or PostgreSQL/MySQL), SQLAlchemy, Pandas

---

Recap: Previous Lesson

- Quick review of file I/O, data structures, or previous data handling modules.
- Introduce the motivation for using SQL with Python (e.g., persistent data storage, querying capabilities).

Using Python Packages to Connect to SQL Databases

We'll use **`sqlite3`** and **`SQLAlchemy`** (high-level ORM) to demonstrate database integration.

Example 1: Using `sqlite3`

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()
cursor.execute('CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY, name TEXT, age INTEGE
conn.commit()
conn.close()
```

Example 2: Using `SQLAlchemy`

```
from sqlalchemy import create_engine, Table, Column, Integer, String, MetaData

engine = create_engine('sqlite:///example.db')
```

```
metadata = MetaData()

users = Table('users', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String),
    Column('age', Integer)
)

metadata.create_all(engine)
```

SQL Statements in Python

We'll cover:

- SELECT
- INSERT
- UPDATE
- DELETE

Example 3: INSERT and SELECT

```
# Using sqlite3
conn = sqlite3.connect('example.db')
cursor = conn.cursor()
cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", ("Alice", 30))
conn.commit()

cursor.execute("SELECT * FROM users")
rows = cursor.fetchall()
print(rows)
conn.close()
```

SQL Predicates, Joins, and Clauses

SQL Predicates:

```
SELECT * FROM users WHERE age > 25;
```

Example 4: INNER JOIN

```
# Assume another table: orders(user_id, product)
cursor.execute('''
    CREATE TABLE IF NOT EXISTS orders (
        id INTEGER PRIMARY KEY,
        user_id INTEGER,
        product TEXT,
        FOREIGN KEY(user_id) REFERENCES users(id)
```

```
    )
''')
conn.commit()

cursor.execute("INSERT INTO orders (user_id, product) VALUES (?, ?)", (1, 'Book'))
conn.commit()

cursor.execute('''
    SELECT users.name, orders.product
    FROM users
    JOIN orders ON users.id = orders.user_id
''')

print(cursor.fetchall())
```

Embedding SQL in Python Functions and Pandas

Example 5: Encapsulating SQL in Functions

```
def get_users_above_age(min_age):
    with sqlite3.connect('example.db') as conn:
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM users WHERE age > ?", (min_age,))
        return cursor.fetchall()
```

Example 6: Using Pandas with SQL

```
import pandas as pd

with sqlite3.connect('example.db') as conn:
    df = pd.read_sql_query("SELECT * FROM users", conn)

print(df)
```

Hands-On Demo: Transactional Update and Result Retrieval

Example 7: Transactional Update

```
def update_user_age(user_id, new_age):
    with sqlite3.connect('example.db') as conn:
        cursor = conn.cursor()
        cursor.execute("UPDATE users SET age = ? WHERE id = ?", (new_age, user_id))
        conn.commit()
        cursor.execute("SELECT * FROM users WHERE id = ?", (user_id,))
        return cursor.fetchall()

print(update_user_age(1, 35))
```

Homework Assignment

> **Goal:** Develop a Python application that connects to a database, performs a series of SQL operations (`SELECT`, `INSERT`, `UPDATE`, `DELETE`), and processes the results with Pandas.

Homework Requirements

- Connect to a database using either `sqlite3` or `SQLAlchemy`.
- Create two tables (e.g., `customers` and `orders`).
- Insert data using Python functions.
- Update and delete records based on conditions.
- Fetch and process results with Pandas.
- Include commentary on how SQL integration benefits data manipulation.

Extra Tips:

- Use transactions and handle exceptions.
- Show intermediate steps using Pandas DataFrames.
- Document your code with comments and/or markdown if using Jupyter Notebook.

Outline2

LESSON 16: INTEGRATING SQL WITH PYTHON

Duration: 2 hours **Objective:** Learn how to integrate SQL with Python for robust, efficient, and persistent data management.

SECTION 1: Introduction and Database Setup

Topics Covered:

- Recap of previous lesson (data handling, files, or structures)
- Introduction to relational databases and SQL
- Using Python to connect with databases

Key Concept:

A database allows structured storage, querying, and updating of large datasets. SQL is the language to interact with databases; Python bridges that interaction.

Tools:

- `sqlite3` (built-in)
- `SQLAlchemy` (optional for abstraction)
- `pandas` (for data manipulation)

Code Demo: Creating a SQLite Database

```
import sqlite3

# Create a new database file
conn = sqlite3.connect('lesson16.db')
cursor = conn.cursor()

# Create a table
cursor.execute('''
CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY,
    name TEXT,
    age INTEGER
)
''')

conn.commit()
conn.close()
```

SECTION 2: Performing SQL Operations in Python

Topics Covered:

- SQL Statements: `SELECT`, `INSERT`, `UPDATE`, `DELETE`
- Predicates and Clauses: `WHERE`, `ORDER BY`, etc.

Explanation:

Each SQL statement can be sent as a string to the database through Python. Parameters should be safely passed using placeholders to avoid SQL injection.

Code Demo: Basic Operations

```
conn = sqlite3.connect('lesson16.db')
cursor = conn.cursor()

# INSERT
cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", ("Alice", 30))

# SELECT
cursor.execute("SELECT * FROM users")
print("All Users:", cursor.fetchall())

# UPDATE
cursor.execute("UPDATE users SET age = ? WHERE name = ?", (31, "Alice"))

# DELETE
```

```
cursor.execute("DELETE FROM users WHERE name = ?", ("Alice",))

conn.commit()
conn.close()
```

SECTION 3: Advanced Queries and Integration with Functions & Pandas

Topics Covered:

- SQL Joins (e.g., INNER JOIN)
- Embedding SQL in reusable Python functions
- Working with query results using Pandas

Code Demo: JOIN and Functions

```
# Create orders table
conn = sqlite3.connect('lesson16.db')
cursor = conn.cursor()
cursor.execute('''
CREATE TABLE IF NOT EXISTS orders (
    id INTEGER PRIMARY KEY,
    user_id INTEGER,
    product TEXT,
    FOREIGN KEY(user_id) REFERENCES users(id)
)
''')

# Insert data
cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", ("Bob", 28))
cursor.execute("INSERT INTO orders (user_id, product) VALUES (?, ?)", (1, "Laptop"))
conn.commit()

# JOIN
cursor.execute('''
SELECT users.name, orders.product
FROM users
JOIN orders ON users.id = orders.user_id
''')

print("Join Result:", cursor.fetchall())
conn.close()
```

Code Demo: Function + Pandas Integration

```
import pandas as pd

def get_users_dataframe():
    with sqlite3.connect('lesson16.db') as conn:
```

```
        return pd.read_sql_query("SELECT * FROM users", conn)

df = get_users_dataframe()
print(df)
```

SECTION 4: Hands-On Practice + Homework Assignment

Hands-On Task: Transactional Update and Fetch

```
def update_age_and_fetch(user_id, new_age):
    with sqlite3.connect('lesson16.db') as conn:
        cursor = conn.cursor()
        cursor.execute("UPDATE users SET age = ? WHERE id = ?", (new_age, user_id))
        conn.commit()
        cursor.execute("SELECT * FROM users WHERE id = ?", (user_id,))
        return cursor.fetchall()

print(update_age_and_fetch(1, 35))
```

Homework Assignment

Objective: Build a complete Python mini-app that connects to a database and demonstrates SQL integration.

Requirements:

1. Connect to a database (SQLite or SQLAlchemy).
2. Create at least two tables (e.g., `customers`, `orders`).
3. Implement:
   - `INSERT`, `SELECT`, `UPDATE`, `DELETE` SQL operations.
   - At least one SQL `JOIN`.
   - One transactional change that is fetched in Python.
4. Use **Pandas** to:
   - Fetch and display the final results.
   - Optionally filter or sort the output.
5. Add **code comments** or markdown cells explaining:
   - Why SQL integration is powerful for structured data.
   - How each SQL command maps to Python logic.

Lesson 17

Exercise

LESSON 17: INTEGRATING SQL WITH PYTHON (Exercise)

Part 1: Core SQL Skills (Assignments 1-10)

1. **Create a Database and Table**
   - Create a SQLite database and a table `employees` with columns: `id`, `name`, `department`, `salary`.
2. **Insert Data into Table**
   - Insert 5 sample records into `employees`.
3. **Basic SELECT Query**
   - Retrieve all records from `employees`.
4. **SELECT with WHERE Clause**
   - Query employees with salary greater than 50000.
5. **Update a Record**
   - Update the salary of an employee with a given `id`.
6. **Delete a Record**
   - Delete an employee with a specific `id`.
7. **ORDER BY Clause**
   - List employees ordered by salary descending.
8. **SELECT with LIKE Operator**
   - Find employees whose name starts with "J".
9. **Aggregate Function - COUNT**
   - Count the total number of employees.
10. **Aggregate Function - AVG with GROUP BY**
    - Find the average salary per department.

Part 2: SQL with Python (sqlite3) (Assignments 11-20)

11. **Connect to SQLite in Python**
    - Write Python code to connect to the database and create a cursor.
12. **Python: Create Table if Not Exists**
    - Use Python to create the `projects` table if it doesn't exist.

13. **Python: Insert Multiple Records with `executemany()`**
    - Insert multiple project records using Python.
14. **Python: SELECT Query and Fetch Results**
    - Write a Python script to fetch and print all employees.
15. **Python: Update Data with Variables**
    - Use Python variables to update a specific employee's salary.
16. **Python: Delete Record with WHERE using Parameters**
    - Delete an employee with a Python parameterized query.
17. **Python: Fetch Results with Filtering**
    - Fetch employees in a specific department using Python.
18. **Handle SQL Exceptions in Python**
    - Implement error handling (try-except) around SQL queries.
19. **Commit Transactions and Close Connection in Python**
    - Practice committing after DML operations and closing the connection.
20. **Bonus Challenge: Create a Simple Report**
    - Write a Python script to list all departments and their total salary expenditure.

Lesson 17- Integrating Sql With Python (Solution)

LESSON 17: INTEGRATING SQL WITH PYTHON (Solution)

Part 1: Core SQL Assignments (1-10)

1. Create Database and Table

```sql
CREATE TABLE employees (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    department TEXT NOT NULL,
    salary REAL
);
```

---

2. Insert Data into Table

```sql
INSERT INTO employees (name, department, salary) VALUES
('John Doe', 'HR', 55000),
('Jane Smith', 'IT', 70000),
('Mike Brown', 'Finance', 48000),
('Sara Wilson', 'IT', 65000),
('Tom Clark', 'HR', 52000);
```

3. SELECT All Records

```
SELECT * FROM employees;
```

---

4. SELECT with WHERE Clause

```
SELECT * FROM employees WHERE salary > 50000;
```

---

5. Update a Record

```
UPDATE employees SET salary = 75000 WHERE id = 2;
```

---

6. Delete a Record

```
DELETE FROM employees WHERE id = 3;
```

---

7. ORDER BY Clause

```
SELECT * FROM employees ORDER BY salary DESC;
```

---

8. SELECT with LIKE Operator

```
SELECT * FROM employees WHERE name LIKE 'J%';
```

---

9. COUNT Aggregate Function

```
SELECT COUNT(*) AS total_employees FROM employees;
```

---

10. AVG with GROUP BY

```
SELECT department, AVG(salary) AS avg_salary
FROM employees
GROUP BY department;
```

Part 2: SQL with Python (sqlite3) Assignments (11-20)

We'll use this standard connection setup for Python assignments:

```
import sqlite3

conn = sqlite3.connect('company.db')
cursor = conn.cursor()
```

---

11. Connect to SQLite in Python

```
import sqlite3

conn = sqlite3.connect('company.db')
cursor = conn.cursor()
print("Connected successfully!")
```

---

12. Create Table if Not Exists

```
cursor.execute('''
CREATE TABLE IF NOT EXISTS projects (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    budget REAL
)
''')
conn.commit()
```

---

13. Insert Multiple Records with `executemany()`

```
projects_data = [
    ('Website Redesign', 10000),
    ('Mobile App', 25000),
    ('Cloud Migration', 15000)
]

cursor.executemany('''
INSERT INTO projects (name, budget) VALUES (?, ?)
''', projects_data)
conn.commit()
```

---

14. SELECT Query and Fetch Results

```
cursor.execute('SELECT * FROM employees')
for row in cursor.fetchall():
    print(row)
```

---

15. Update Data with Variables

```
emp_id = 1
new_salary = 60000
cursor.execute('''
UPDATE employees SET salary = ? WHERE id = ?
''', (new_salary, emp_id))
conn.commit()
```

---

16. Delete Record with WHERE using Parameters

```
emp_id_to_delete = 5
cursor.execute('DELETE FROM employees WHERE id = ?', (emp_id_to_delete,))
conn.commit()
```

---

17. Fetch Results with Filtering

```
department = 'IT'
cursor.execute('SELECT * FROM employees WHERE department = ?', (department,))
for row in cursor.fetchall():
    print(row)
```

---

18. Handle SQL Exceptions in Python

```
try:
    cursor.execute('SELECT * FROM non_existing_table')
except sqlite3.Error as e:
    print("An error occurred:", e)
```

---

19. Commit Transactions and Close Connection

```
conn.commit()
conn.close()
print("Connection closed.")
```

---

20. Bonus Challenge: Create a Simple Report

```
cursor.execute('''
SELECT department, SUM(salary) as total_salary
FROM employees
GROUP BY department
''')
for row in cursor.fetchall():
    print(f"Department: {row[0]}, Total Salary: {row[1]}")
```

Lesson 18

Ii. Creating Layers And Adding Geometrics To Data

II. Creating Layers and Adding Geometrics to Data

In Matplotlib, each element you add to a chart — such as points, lines, labels, grids, or shapes — acts like a **layer**. These layers stack together to build the final visual. Understanding how to combine these layers gives you flexibility to create clear, insightful charts.

Data

```
import pandas as pd

data = {
    'Model': ['Alpha', 'Beta', 'Gamma', 'Delta', 'Epsilon', 'Zeta', 'Eta', 'Theta', 'Iota', 'Ka
    'Price': [15000, 18000, 20000, 22000, 25000, 27000, 30000, 32000, 35000, 40000],
    'Sales': [800, 750, 500, 450, 400, 380, 300, 250, 200, 150],
    'Rating': [4.5, 4.2, 4.8, 4.0, 3.9, 3.8, 3.5, 3.4, 3.2, 3.0]
}

df = pd.DataFrame(data)
```

Basic Line Plot (Price vs Sales)

A line plot connects data points with straight lines. It is typically used to show trends or relationships between continuous variables.

```
import matplotlib.pyplot as plt

plt.plot(df['Price'], df['Sales'], label='Sales Trend')
plt.title('Car Price vs Sales')
plt.xlabel('Price')
plt.ylabel('Sales')
plt.legend()
```

```
plt.grid(True)  # Adding a grid as a background layer
plt.show()
```

Explanation:

- `plt.plot()` adds a **line layer**.
- `plt.title()`, `plt.xlabel()`, and `plt.ylabel()` add **text layers**.
- `plt.legend()` adds a **legend layer** for clarity.
- `plt.grid(True)` adds a **grid layer** beneath the data.

These layers combine to make the chart readable and informative.

Scatter Plot (Rating vs Sales)

Scatter plots are used to show the relationship between two numerical variables. Each point is an observation.

```
plt.scatter(df['Rating'], df['Sales'], color='green', alpha=0.7, label='Data Points')
plt.title('Rating vs Sales')
plt.xlabel('Rating')
plt.ylabel('Sales')
plt.legend()
plt.grid(True)
plt.show()
```

What's Happening Here:

- `plt.scatter()` adds a **point layer**.
- The `alpha` parameter controls transparency, so overlapping points are visible.
- The legend and grid are layered on top for readability.

Combining Layers — Scatter with Trend Line

Often, it's useful to combine a scatter plot with a line of best fit to highlight a trend.

```
import numpy as np

# Scatter plot
plt.scatter(df['Rating'], df['Sales'], color='blue', alpha=0.6, label='Sales Data')

# Trend line (linear fit)
coefficients = np.polyfit(df['Rating'], df['Sales'], 1)
poly = np.poly1d(coefficients)
plt.plot(df['Rating'], poly(df['Rating']), color='red', linewidth=2, label='Trend Line')

plt.title('Rating vs Sales with Trend Line')
plt.xlabel('Rating')
plt.ylabel('Sales')
plt.legend()
```

```
plt.grid(True)
plt.show()
```

Why This Works:

- The **scatter layer** shows actual data.
- The **trend line layer** highlights the overall relationship.
- Using layers together helps the viewer see both individual points and general trends.

Highlighting Data — Using Annotations

Annotations add another layer of meaning to a chart by pointing out key data.

```
plt.scatter(df['Price'], df['Sales'], color='purple', alpha=0.7)
plt.title('Car Price vs Sales with Annotation')
plt.xlabel('Price')
plt.ylabel('Sales')
plt.grid(True)

# Annotate a specific point
plt.annotate('Best Seller', xy=(20000, 500), xytext=(22000, 600),
             arrowprops=dict(facecolor='black', arrowstyle='->'))

plt.show()
```

Reasoning:

- `plt.annotate()` adds a **text + arrow layer** that draws attention to important data.
- Use annotations to make insights stand out in presentations or reports.

Summary of Layering Concepts in Matplotlib

Layer Type	Example Function	Purpose
Plot Line	`plt.plot()`	Show trends between variables
Scatter Points	`plt.scatter()`	Show individual data points
Title/Labels	`plt.title()`, `plt.xlabel()`	Add context
Legend	`plt.legend()`	Identify layers
Grid	`plt.grid(True)`	Make values easier to read
Annotation	`plt.annotate()`	Highlight key insights

Iii. Understanding Types Of Graphs For Types Of Variables

III. Understanding Types of Graphs for Types of Variables

Choosing the right type of graph depends on the nature of the variables you are analyzing. Here's a quick summary:

Variable Type	Best Graph Type	Example
Continuous vs Continuous	Scatter Plot	Rating vs Sales
Categorical vs Continuous	Bar Plot	Type vs Average Price
Single Continuous	Histogram	Distribution of Prices
Single Categorical	Count Plot	Count of Car Types

Extended Dataset with Extra Columns

We will add a `Type` column (car category) and a `Country` column (origin), commonly used for categorical comparisons.

```
import pandas as pd

data = {
    'Model': ['Alpha', 'Beta', 'Gamma', 'Delta', 'Epsilon', 'Zeta', 'Eta', 'Theta', 'Iota', 'Ka
    'Price': [15000, 18000, 20000, 22000, 25000, 27000, 30000, 32000, 35000, 40000],
    'Sales': [800, 750, 500, 450, 400, 380, 300, 250, 200, 150],
    'Rating': [4.5, 4.2, 4.8, 4.0, 3.9, 3.8, 3.5, 3.4, 3.2, 3.0],
    'Type': ['Hatchback', 'Hatchback', 'Sedan', 'Sedan', 'SUV', 'SUV', 'SUV', 'Truck', 'Truck'
    'Country': ['Japan', 'Japan', 'Germany', 'Germany', 'USA', 'USA', 'USA', 'USA', 'Japan', '
}

df = pd.DataFrame(data)
```

A) Continuous vs Continuous — Scatter Plot

Used to explore the relationship between two continuous variables.

```
import matplotlib.pyplot as plt

plt.scatter(df['Rating'], df['Sales'], color='teal', alpha=0.7)
plt.title('Rating vs Sales')
plt.xlabel('Rating')
plt.ylabel('Sales')
plt.grid(True)
plt.show()
```

Use When:

- Both variables are numerical.
- You want to see trends, patterns, or correlations.

B) Categorical vs Continuous — Bar Plot (Type vs Average Price)

Compare means or aggregates across categories.

```
avg_price = df.groupby('Type')['Price'].mean()

avg_price.plot(kind='bar', color='skyblue')
plt.title('Average Price by Car Type')
plt.xlabel('Car Type')
plt.ylabel('Average Price')
plt.grid(axis='y')
plt.show()
```

Alternative with Matplotlib:

```
plt.bar(avg_price.index, avg_price.values, color='coral')
plt.title('Average Price by Car Type')
plt.xlabel('Car Type')
plt.ylabel('Average Price')
plt.grid(axis='y')
plt.show()
```

Use When:

- You want to compare numerical summaries (mean, sum) between categories.

C) Single Continuous Variable — Histogram (Price Distribution)

Visualize the distribution of a continuous variable.

```
plt.hist(df['Price'], bins=5, color='purple', edgecolor='black')
plt.title('Price Distribution')
plt.xlabel('Price')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()
```

Use When:

- You want to see the spread and shape of data (e.g., normal distribution, skewness).

D) Single Categorical Variable — Count Plot (Car Type Frequency)

Show how many items fall into each category.

```
counts = df['Type'].value_counts()

plt.bar(counts.index, counts.values, color='orange')
plt.title('Count of Car Types')
plt.xlabel('Car Type')
plt.ylabel('Count')
plt.grid(axis='y')
plt.show()
```

Use When:

- You want to show how frequently each category appears in your dataset.

E) Grouped Bar Plot — Average Sales by Country

When comparing multiple categories side by side.

```
avg_sales_country = df.groupby('Country')['Sales'].mean()

plt.bar(avg_sales_country.index, avg_sales_country.values, color='green')
plt.title('Average Sales by Country')
plt.xlabel('Country')
plt.ylabel('Average Sales')
plt.grid(axis='y')
plt.show()
```

F) Box Plot — Sales Distribution by Type

Box plots are excellent for visualizing distribution and outliers per category.

```
types = df['Type'].unique()
types.sort()
data_to_plot = [df[df['Type'] == t]['Price'] for t in types]


plt.figure(figsize=(8, 6))
plt.boxplot(data_to_plot, labels=types)
plt.title("Price Distribution by Vehicle Type")
plt.xlabel("Type")
plt.ylabel("Price")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()


# of with seaborn
import seaborn as sns

sns.boxplot(x='Type', y='Sales', data=df)
plt.title('Sales Distribution by Car Type')
plt.show()
```

Use When:

- You want to check spread, median, and outliers by category.

Summary of Graph Choices and When to Use Them

Graph Type	Best For	Typical Variables
Scatter Plot	Trends/Correlation	Continuous vs Continuous
Bar Plot	Compare Categories	Categorical vs Continuous
Histogram	Distribution Analysis	Single Continuous
Count Plot	Category Frequency	Single Categorical
Box Plot	Distribution + Outliers	Categorical vs Continuous
Grouped Bar	Comparing Categories by Group	Categorical grouped

---

Iv. Adding Text, Grid, Lines, And Legends To Graphs

IV. Adding Text, Grid, Lines, and Legends to Graphs

Enhancing your plots with extra elements improves their clarity and visual appeal. Let's go through them one by one:

A) Adding Titles and Axis Labels

These provide context and help the viewer understand what the chart shows.

```
plt.scatter(df['Price'], df['Sales'], color='green')
plt.title('Car Price vs Sales')  # Title at the top
plt.xlabel('Price (USD)')        # X-axis label
plt.ylabel('Sales (Units)')      # Y-axis label
plt.show()
```

B) Adding Gridlines

Grids make reading values easier, especially on scatter plots and bar charts.

```
plt.scatter(df['Price'], df['Sales'], color='blue')
plt.title('Car Price vs Sales with Grid')
plt.xlabel('Price')
plt.ylabel('Sales')
plt.grid(True)  # Add default gridlines
plt.show()
```

Customize Gridlines:

```
plt.scatter(df['Price'], df['Sales'], color='blue')
plt.title('Car Price vs Sales with Custom Grid')
plt.xlabel('Price')
plt.ylabel('Sales')
plt.grid(color='gray', linestyle='--', linewidth=0.5)  # Light dashed grid
plt.show()
```

C) Adding Reference Lines (axhline, axvline)

Use these to show thresholds, averages, or targets.

```
plt.scatter(df['Price'], df['Sales'], color='purple')
plt.title('Car Price vs Sales with Reference Line')
plt.xlabel('Price')
plt.ylabel('Sales')
plt.grid(True)

# Add horizontal line for average sales
avg_sales = df['Sales'].mean()
plt.axhline(y=avg_sales, color='red', linestyle='--', label=f'Average Sales ({avg_sales:.0f})'

plt.legend()
plt.show()
```

Vertical Line Example (e.g., Median Price):

```
plt.scatter(df['Price'], df['Sales'], color='orange')
plt.title('Car Price vs Sales with Median Price Line')
plt.xlabel('Price')
plt.ylabel('Sales')
plt.grid(True)

median_price = df['Price'].median()
plt.axvline(x=median_price, color='green', linestyle='-.', label=f'Median Price (${median_price

plt.legend()
plt.show()
```

D) Adding Text Labels on Points

Annotate each data point for identification.

```
plt.scatter(df['Price'], df['Sales'], color='navy')
plt.title('Car Price vs Sales with Model Labels')
plt.xlabel('Price')
plt.ylabel('Sales')
plt.grid(True)

for i in range(len(df)):
    plt.text(df['Price'][i]+200, df['Sales'][i]+5, df['Model'][i], fontsize=9)

plt.show()
```

E) Adding Annotations with Arrows

Use `plt.annotate()` to highlight special points with arrows.

```
plt.scatter(df['Price'], df['Sales'], color='teal')
plt.title('Car Price vs Sales with Highlighted Best Seller')
plt.xlabel('Price')
plt.ylabel('Sales')
plt.grid(True)

# Annotate "Gamma" model
best_seller = df[df['Model'] == 'Gamma']
plt.annotate('Best Seller',
             xy=(best_seller['Price'].values[0], best_seller['Sales'].values[0]),
             xytext=(best_seller['Price'].values[0]+5000, best_seller['Sales'].values[0]+100),
             arrowprops=dict(facecolor='black', arrowstyle='->'))

plt.show()
```

F) Adding Legends

Legends explain colors, lines, or markers used in the chart.

```
plt.scatter(df['Price'], df['Sales'], color='blue', label='Sales Data')
plt.axhline(y=avg_sales, color='red', linestyle='--', label='Average Sales')
plt.title('Car Price vs Sales with Legend')
plt.xlabel('Price')
plt.ylabel('Sales')
plt.grid(True)
plt.legend()  # Show the legend box
plt.show()
```

G) Combining Multiple Elements in One Plot

```
plt.scatter(df['Price'], df['Sales'], color='darkgreen', label='Sales Data')
plt.axhline(y=avg_sales, color='red', linestyle='--', label='Average Sales')
plt.axvline(x=median_price, color='orange', linestyle='-.', label='Median Price')

for i in range(len(df)):
    plt.text(df['Price'][i]+200, df['Sales'][i]+5, df['Model'][i], fontsize=8)

plt.title('Car Price vs Sales - Fully Annotated')
plt.xlabel('Price')
plt.ylabel('Sales')
plt.grid(True)
plt.legend()
plt.show()
```

Summary of Elements You Can Add:

Element	Function	Command
Title	Adds chart title	`plt.title()`
Axis Labels	Names X and Y axes	`plt.xlabel()`, `plt.ylabel()`
Grid	Background lines for readability	`plt.grid()`
Reference Line	Shows thresholds or averages	`plt.axhline()`, `plt.axvline()`
Text Labels	Adds text to specific points	`plt.text()`
Annotations	Highlights with arrows	`plt.annotate()`
Legend	Explains data elements	`plt.legend()`

Lesson18

Introduction to Matplotlib in Python

What is Matplotlib?

Matplotlib is one of the most popular Python libraries for creating **static**, **animated**, and **interactive** visualizations. It's highly customizable and works well with other Python libraries like NumPy, pandas, and SciPy.

Installation

If not already installed, you can install Matplotlib via pip:

```
pip install matplotlib
```

Importing Matplotlib

We typically import the `pyplot` submodule for most plotting functions.

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

Sample Dataset

Let's create a simple dataset to use across demos:

```
# Create sample dataset
data = {
```

```
    'Month': ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun'],
    'Sales': [305, 356, 410, 489, 512, 590],
    'Expenses': [220, 260, 300, 320, 340, 390]
}

df = pd.DataFrame(data)
df['Price'] = df['Sales'] - df['Expenses']
```

Basic Plot Types

1. Line Plot

```
plt.plot(df['Month'], df['Sales'], label='Sales')
plt.plot(df['Month'], df['Expenses'], label='Expenses')
plt.title('Monthly Sales vs Expenses')
plt.xlabel('Month')
plt.ylabel('Amount ($)')
plt.legend()
plt.grid(True)
plt.show()
```

---

2. Bar Chart

```
plt.bar(df['Month'], df['Sales'], color='skyblue')
plt.title('Monthly Sales')
plt.xlabel('Month')
plt.ylabel('Sales ($)')
plt.show()
```

---

3. Stacked Bar Chart

```
plt.bar(df['Month'], df['Sales'], label='Sales')
plt.bar(df['Month'], df['Expenses'], bottom=df['Sales'], label='Expenses')
plt.title('Stacked Sales & Expenses')
plt.legend()
plt.show()
```

---

4. Scatter Plot

```
plt.scatter(df['Sales'], df['Expenses'], color='red')
plt.title('Sales vs Expenses')
plt.xlabel('Sales')
plt.ylabel('Expenses')
```

```
plt.grid(True)
plt.show()
```

---

5. Pie Chart

```
plt.pie(df['Sales'], labels=df['Month'], autopct='%1.1f%%')
plt.title('Sales Distribution by Month')
plt.show()
```

Customization Options

Titles, Labels, Legends, and Grids

```
plt.plot(df['Month'], df['Sales'], color='green', linestyle='--', marker='o')
plt.title('Sales Trend')
plt.xlabel('Month')
plt.ylabel('Sales')
plt.legend(['Sales'])
plt.grid(True, linestyle=':', linewidth=0.5)
plt.show()
```

Subplots

```
fig, axs = plt.subplots(2, 1, figsize=(8, 6))
axs[0].bar(df['Month'], df['Sales'], color='blue')
axs[0].set_title('Sales')
axs[1].bar(df['Month'], df['Expenses'], color='orange')
axs[1].set_title('Expenses')
plt.tight_layout()
plt.show()
```

Styling with `plt.style`

```
plt.style.use('ggplot')
plt.plot(df['Month'], df['Sales'])
plt.title('Styled Plot')
plt.show()
```

Advanced Features

Plotting with NumPy

```
x = np.linspace(0, 10, 100)
y = np.sin(x)

plt.plot(x, y)
plt.title('Sine Wave')
plt.xlabel('x')
plt.ylabel('sin(x)')
plt.grid(True)
plt.show()
```

Annotating Points

```
plt.plot(df['Month'], df['Sales'], marker='o')
plt.title('Monthly Sales with Annotations')

for i, txt in enumerate(df['Sales']):
    plt.annotate(txt, (df['Month'][i], df['Sales'][i]), textcoords="offset points", xytext=(0,5
plt.show()
```

Histogram

```
np.random.seed(42)
data = np.random.randn(1000)


plt.hist(data, bins=30, color='purple', alpha=0.7)
plt.title('Histogram of Random Data')
plt.show()
```

Saving Plots

```
plt.plot(df['Month'], df['Sales'])
plt.title('Sales')
plt.savefig('sales_plot.png', dpi=300)
plt.show()
```

Interactive Plots (Optional with `%matplotlib notebook` or `matplotlib.widgets`)

```
# In a Jupyter Notebook:
%matplotlib notebook
```

```
plt.plot(df['Month'], df['Sales'])
plt.title("Interactive Plot")
plt.show()
```

Resetting the Plot

```
plt.clf()       # Clears the current figure
plt.cla()       # Clears the current axes
plt.close()     # Closes the figure window
```

Summary of Key Components

Feature	Function
Basic Plot	`plt.plot()`
Labels	`plt.xlabel()`, `plt.ylabel()`
Title	`plt.title()`
Legend	`plt.legend()`
Grid	`plt.grid()`
Subplots	`plt.subplots()`
Save Plot	`plt.savefig()`
Pie/Bar/Hist	`plt.pie()`, `plt.bar()`, `plt.hist()`

V. Extended Dataset With Date And Extra Models

V. Extended Dataset with Date and Extra Models

```
import pandas as pd

data = {
    'Model': ['Alpha', 'Beta', 'Gamma', 'Delta', 'Epsilon', 'Zeta', 'Eta', 'Theta', 'Iota', 'Ka
    'Price': [15000, 18000, 20000, 22000, 25000, 27000, 30000, 32000, 35000, 40000, 28000, 230
    'Sales': [800, 750, 500, 450, 400, 380, 300, 250, 200, 150, 360, 420],
    'Rating': [4.5, 4.2, 4.8, 4.0, 3.9, 3.8, 3.5, 3.4, 3.2, 3.0, 3.6, 4.1],
    'Type': ['Hatchback', 'Hatchback', 'Sedan', 'Sedan', 'SUV', 'SUV', 'SUV', 'Truck', 'Truck'
    'Country': ['Japan', 'Japan', 'Germany', 'Germany', 'USA', 'USA', 'USA', 'USA', 'Japan', '(
    'LaunchDate': pd.to_datetime(['2021-01-01', '2021-03-01', '2021-05-01', '2021-06-01',
                                  '2021-07-01', '2021-08-01', '2021-09-01', '2021-10-01',
                                  '2021-11-01', '2021-12-01', '2022-01-01', '2022-02-01'])
}

df = pd.DataFrame(data)
```

Visual Examples Covering All Graph Types & Layers

---

1) Line Plot — Price over Time

```
import matplotlib.pyplot as plt

plt.plot(df['LaunchDate'], df['Price'], marker='o', color='blue', label='Price Trend')
plt.title('Price Over Time')
plt.xlabel('Launch Date')
plt.ylabel('Price')
plt.grid(True)
plt.legend()
plt.show()
```

---

2) Scatter Plot with Trend Line — Rating vs Sales

```
import numpy as np

plt.scatter(df['Rating'], df['Sales'], color='purple', label='Data Points')

# Trend Line (Linear Fit)
coeff = np.polyfit(df['Rating'], df['Sales'], 1)
poly_eqn = np.poly1d(coeff)
plt.plot(df['Rating'], poly_eqn(df['Rating']), color='red', linewidth=2, label='Trend Line')

plt.title('Rating vs Sales with Trend Line')
plt.xlabel('Rating')
plt.ylabel('Sales')
plt.grid(True)
plt.legend()
plt.show()
```

---

3) Bar Plot — Average Price by Type

```
avg_price_type = df.groupby('Type')['Price'].mean()

plt.bar(avg_price_type.index, avg_price_type.values, color='skyblue')
plt.title('Average Price by Car Type')
plt.xlabel('Car Type')
plt.ylabel('Average Price')
plt.grid(axis='y')
plt.show()
```

---

4) Histogram — Sales Distribution

```
plt.hist(df['Sales'], bins=5, color='green', edgecolor='black')
plt.title('Sales Distribution')
plt.xlabel('Sales')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()
```

---

5) Count Plot (Bar Plot) — Count of Models per Country

```
country_counts = df['Country'].value_counts()

plt.bar(country_counts.index, country_counts.values, color='orange')
plt.title('Number of Models by Country')
plt.xlabel('Country')
plt.ylabel('Count')
plt.grid(axis='y')
plt.show()
```

---

6) Box Plot — Price Distribution by Type (Using Seaborn)

```
import seaborn as sns

sns.boxplot(x='Type', y='Price', data=df)
plt.title('Price Distribution by Car Type')
plt.grid(True)
plt.show()
```

---

7) Time Series Line Chart — Sales over Time with Annotation

```
plt.plot(df['LaunchDate'], df['Sales'], marker='o', label='Sales Over Time')
plt.title('Sales Over Time with Highlight')
plt.xlabel('Launch Date')
plt.ylabel('Sales')
plt.grid(True)

# Highlight the highest sales point
best = df[df['Sales'] == df['Sales'].max()]
plt.annotate('Best Sales', xy=(best['LaunchDate'].values[0], best['Sales'].values[0]),
             xytext=(best['LaunchDate'].values[0], best['Sales'].values[0]+100),
             arrowprops=dict(facecolor='black', arrowstyle='->'))

plt.legend()
plt.show()
```

---

8) Pie Chart — Market Share by Country

```
country_sales = df.groupby('Country')['Sales'].sum()

plt.pie(country_sales.values, labels=country_sales.index, autopct='%1.1f%%', startangle=140)
plt.title('Sales Market Share by Country')
plt.show()
```

---

9) Multi-Layer Plot — Sales vs Price Colored by Type

```
colors = {'Hatchback':'blue', 'Sedan':'green', 'SUV':'red', 'Truck':'purple'}

plt.figure(figsize=(8,6))
for t in df['Type'].unique():
    subset = df[df['Type'] == t]
    plt.scatter(subset['Price'], subset['Sales'], color=colors[t], label=t)

plt.title('Sales vs Price by Car Type')
plt.xlabel('Price')
plt.ylabel('Sales')
plt.grid(True)
plt.legend()
plt.show()
```

---

10) Correlation Heatmap — Numeric Variables

```
numeric_df = df.select_dtypes(include='number')
corr = numeric_df.corr().values
labels = numeric_df.columns


fig, ax = plt.subplots(figsize=(8, 6))
cax = ax.matshow(corr, cmap='coolwarm')
fig.colorbar(cax)


ax.set_xticks(np.arange(len(labels)))
ax.set_yticks(np.arange(len(labels)))
ax.set_xticklabels(labels, rotation=45, ha='left')
ax.set_yticklabels(labels)


# Annotate correlation values
for i in range(len(labels)):
```

```
    for j in range(len(labels)):
        ax.text(j, i, f"{corr[i, j]:.2f}", va='center', ha='center', color='black')
plt.title("Correlation Heatmap", pad=20)
plt.tight_layout()
plt.show()

# or with seaborn


sns.heatmap(df[['Price', 'Sales', 'Rating']].corr(), annot=True, cmap='coolwarm')
plt.title('Correlation Heatmap')
plt.show()
```

Reusable Plotting Function Example

```
def plot_with_reference(df, x, y, title='', xlabel='', ylabel='', reference=None, annotate=Fals
    plt.scatter(df[x], df[y], color='teal', label='Data Points')
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.grid(True)

    if reference == 'mean':
        mean_val = df[y].mean()
        plt.axhline(y=mean_val, color='red', linestyle='--', label=f'Mean {y}: {mean_val:.2f}'

    if annotate:
        for i in range(len(df)):
            plt.text(df[x].iloc[i], df[y].iloc[i]+10, df['Model'].iloc[i], fontsize=8)

    plt.legend()
    plt.show()
```

Example Use:

```
plot_with_reference(df, 'Price', 'Sales',
                    title='Price vs Sales with Mean Line and Annotations',
                    xlabel='Price', ylabel='Sales',
                    reference='mean', annotate=True)
```

Modified Multi-Scatter Plot with Reference and Annotations

```
def multi_plot_with_reference(df, x_vars, y, colors, title='', xlabel='', ylabel='', reference=
    plt.figure(figsize=(8,6))

    for i, x in enumerate(x_vars):
```

```
        plt.scatter(df[x], df[y], color=colors[i], label=f'{x} vs {y}')

        if reference == 'mean':
            mean_val = df[y].mean()
            plt.axhline(y=mean_val, color='red', linestyle='--', label=f'Mean {y}: {mean_val:.

        if annotate:
            for j in range(len(df)):
                plt.text(df[x].iloc[j], df[y].iloc[j]+10, df['Model'].iloc[j], fontsize=8, colo

    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.grid(True)
    plt.legend()
    plt.show()
```

Plotting Both — Price vs Sales and Rating vs Sales Together

```
multi_plot_with_reference(
    df,
    x_vars=['Price', 'Rating'],
    y='Sales',
    colors=['blue', 'green'],
    title='Price & Rating vs Sales with Mean Line and Annotations',
    xlabel='Price / Rating',
    ylabel='Sales',
    reference='mean',
    annotate=True
)
```

What This Does:

- Plots **Price vs Sales** in **blue**
- Plots **Rating vs Sales** in **green**
- Adds the **mean Sales reference line** in **red**
- Annotates all points with model names in corresponding colors
- Includes a legend to differentiate the two series

Please note:

- Because **Price** and **Rating** have different scales, this works as an **exploratory visual**, but if you want a more accurate comparison, you may consider:
  - Scaling variables (e.g., min-max normalization)

- Using twin axes with `plt.twinx()`
- Creating separate subplots

Summary of What's Covered

Graph Type	Example
Line Plot	Price Over Time
Scatter Plot + Trend	Rating vs Sales
Bar Plot	Average Price by Type
Histogram	Sales Distribution
Count Plot	Models per Country
Box Plot	Price by Type
Time Series + Annotation	Sales Over Time
Pie Chart	Market Share by Country
Multi-Layer Scatter	Sales vs Price by Type
Heatmap	Correlation Matrix
Custom Function	Reusable Scatter + Reference

Or

Summary of Graph Types & Use-Cases

Graph Type	Use Case
Line Plot	Trends over time
Scatter Plot	Correlation between variables
Bar Chart	Compare categories
Histogram	Distribution of a variable
Box Plot	Summary statistics and outliers
Pie Chart	Proportion of categories

Conclusion

With this extended dataset and examples, you've got:

- Different variable combinations
- Visual layering (trend lines, annotations, reference lines)
- Statistical plots and summaries
- Time series and categorical data handling
- Reusable plotting patterns for your lessons

Vi. Hands-On + Homework

VI. Hands-on: Working with Data to Prepare 3 Different Plots

Your task during this hands-on session is to practice combining the plotting skills you've learned. You'll work with our car dataset (the extended one from this lesson) and create the following plots.

A) Line Plot — Sales Trend Over Time

```
plt.plot(df['LaunchDate'], df['Sales'], marker='o', color='blue')
plt.title('Sales Trend Over Launch Dates')
plt.xlabel('Launch Date')
plt.ylabel('Sales')
plt.grid(True)

# Add average sales reference line
plt.axhline(y=df['Sales'].mean(), color='red', linestyle='--', label='Average Sales')
plt.legend()
plt.show()
```

** Practice Focus:**

- Time-series plotting
- Adding reference lines and legends
- Using dates as x-axis

B) Box Plot — Price Distribution by Car Type

```
sns.boxplot(x='Type', y='Price', data=df)
plt.title('Price Distribution by Car Type')
plt.grid(True)
plt.show()
```

** Practice Focus:**

- Category-wise distribution
- Understanding spread and outliers
- Using Seaborn with Matplotlib

C) Pie Chart — Market Share of Car Types

```
type_counts = df['Type'].value_counts()
plt.pie(type_counts, labels=type_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Market Share by Car Type')
plt.show()
```

Practice Focus:

- Visualizing proportions
- Customizing pie charts (start angle, labels)

D) BONUS Example — Combined Scatter + Reference + Annotation

```
plt.scatter(df['Price'], df['Sales'], color='green', label='Sales Data')
plt.title('Price vs Sales with Annotation')
plt.xlabel('Price')
plt.ylabel('Sales')
plt.grid(True)

# Reference line for average sales
plt.axhline(y=df['Sales'].mean(), color='red', linestyle='--', label='Average Sales')

# Annotate highest sale
best = df[df['Sales'] == df['Sales'].max()]
plt.annotate('Top Model', xy=(best['Price'].values[0], best['Sales'].values[0]),
             xytext=(best['Price'].values[0]+5000, best['Sales'].values[0]+100),
             arrowprops=dict(facecolor='black', arrowstyle='->'))

plt.legend()
plt.show()
```

** Practice Focus:**

- Combining scatter, reference lines, annotations, and legend

---

VII. Homework Assignment (Optional or To Continue at Home)

Using **any dataset you have** (or the provided car dataset), prepare the following visualizations:

1. Histogram of a Continuous Variable (e.g., Price or Sales)

```
plt.hist(df['Price'], bins=6, color='orange', edgecolor='black')
plt.title('Price Distribution')
plt.xlabel('Price')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()
```

---

2. Bar Chart Comparing Categories (e.g., Car Type vs Average Sales)

```
avg_sales_by_type = df.groupby('Type')['Sales'].mean()

plt.bar(avg_sales_by_type.index, avg_sales_by_type.values, color='purple')
plt.title('Average Sales by Car Type')
plt.xlabel('Car Type')
plt.ylabel('Average Sales')
plt.grid(axis='y')
plt.show()
```

---

3. Scatter Plot of Two Continuous Variables (e.g., Price vs Sales) with Annotations

```
plt.scatter(df['Price'], df['Sales'], color='navy')
plt.title('Price vs Sales')
plt.xlabel('Price')
plt.ylabel('Sales')
plt.grid(True)

for i in range(len(df)):
    plt.text(df['Price'][i]+200, df['Sales'][i]+5, df['Model'][i], fontsize=8)

plt.show()
```

Bonus Challenge:

- Create a function similar to `plot_with_reference()` that allows plotting **any two variables** with optional annotations and reference lines.
- Apply it to your dataset with different pairs of variables.

```
fig, ax1 = plt.subplots(figsize=(10, 6))


color = 'tab:blue'
ax1.set_xlabel('Model')
ax1.set_ylabel('Price', color=color)
ax1.plot(df['Model'], df['Price'], marker='o', color=color, label='Price')
ax1.tick_params(axis='y', labelcolor=color)
ax1.set_xticklabels(df['Model'], rotation=45)


ax2 = ax1.twinx()
color = 'tab:red'
ax2.set_ylabel('Sales', color=color)
ax2.plot(df['Model'], df['Sales'], marker='s', color=color, label='Sales')
ax2.tick_params(axis='y', labelcolor=color)


fig.tight_layout()
plt.title("Price vs Sales by Model")
plt.show()
```

Lesson 19

Ensuring Plots Display Well In This Environment

The solutions for your **Matplotlib assignments on the extended dataset (1000 rows)** have been prepared and plotted with realistic, readable results.

Here's a summary of what was done:

- **Grouped Line and Bar Charts** for aggregated values like sales and price per model.
- **Scatter Plots** showing relationships, with clear labels and adjusted transparency.
- **Histograms** and **Boxplots** visualizing distributions.
- **Pie Charts** to display category proportions.
- **Time-Series with Cumulative and Rolling Trends**, showing growth and smoothed patterns.
- **Dual Axis Plot** for comparing price and sales over time.
- **Subplots** with mixed chart types for comprehensive dashboards.

```
import matplotlib.pyplot as plt

# Ensuring plots display well in this environment
plt.rcParams.update({'figure.max_open_warning': 0})

# Creating Total Revenue and Profit Margin columns for later tasks
df_large['TotalRevenue'] = df_large['Price'] * df_large['Sales']
df_large['ProfitMargin'] = df_large['TotalRevenue'] / df_large['Price']

# Rolling average on Sales (for assignment #22)
df_large_sorted = df_large.sort_values('LaunchDate').reset_index(drop=True)
df_large_sorted['RollingSales'] = df_large_sorted['Sales'].rolling(window=20).mean()

# Cumulative Sales and Revenue (for assignment #20 and #24)
df_large_sorted['CumulativeSales'] = df_large_sorted['Sales'].cumsum()
df_large_sorted['CumulativeRevenue'] = df_large_sorted['TotalRevenue'].cumsum()

# Storing key plots in a list for better handling
plots_summary = []

# Example 1 - Line Plot of Sales over Models (grouped by Model)
sales_by_model = df_large.groupby('Model')['Sales'].sum().sort_values()
fig1, ax1 = plt.subplots()
ax1.plot(sales_by_model.index, sales_by_model.values, marker='o')
ax1.set_title('Total Sales per Model')
ax1.set_xlabel('Model')
ax1.set_ylabel('Total Sales')
ax1.grid(True)
plots_summary.append(('Line Plot - Total Sales per Model', fig1))

# Example 2 - Bar Chart of Average Price per Model
avg_price_model = df_large.groupby('Model')['Price'].mean().sort_values()
fig2, ax2 = plt.subplots()
ax2.bar(avg_price_model.index, avg_price_model.values)
ax2.set_title('Average Price per Model')
ax2.set_xlabel('Model')
ax2.set_ylabel('Average Price ($)')
plt.xticks(rotation=45)
```

```
plots_summary.append(('Bar Chart - Average Price per Model', fig2))

# Example 3 - Scatter Plot Price vs Sales
fig3, ax3 = plt.subplots()
ax3.scatter(df_large['Price'], df_large['Sales'], alpha=0.5)
ax3.set_title('Price vs Sales')
ax3.set_xlabel('Price ($)')
ax3.set_ylabel('Sales')
ax3.grid(True)
plots_summary.append(('Scatter Plot - Price vs Sales', fig3))

# Example 4 - Histogram of Ratings
fig4, ax4 = plt.subplots()
ax4.hist(df_large['Rating'], bins=10, color='skyblue', edgecolor='black')
ax4.set_title('Distribution of Ratings')
ax4.set_xlabel('Rating')
ax4.set_ylabel('Frequency')
plots_summary.append(('Histogram - Distribution of Ratings', fig4))

# Example 5 - Pie Chart of Vehicle Types
type_counts = df_large['Type'].value_counts()
fig5, ax5 = plt.subplots()
ax5.pie(type_counts, labels=type_counts.index, autopct='%1.1f%%', startangle=90)
ax5.set_title('Vehicle Type Distribution')
plots_summary.append(('Pie Chart - Vehicle Type Distribution', fig5))

# Example 6 - Cumulative Sales over Time
fig6, ax6 = plt.subplots()
ax6.plot(df_large_sorted['LaunchDate'], df_large_sorted['CumulativeSales'])
ax6.set_title('Cumulative Sales Over Time')
ax6.set_xlabel('Launch Date')
ax6.set_ylabel('Cumulative Sales')
ax6.grid(True)
plots_summary.append(('Line Plot - Cumulative Sales Over Time', fig6))

# Example 7 - Dual Axis Line Plot: Price and Sales Over Time
fig7, ax7 = plt.subplots()
ax7.plot(df_large_sorted['LaunchDate'], df_large_sorted['Price'], 'g-', label='Price')
ax8 = ax7.twinx()
ax8.plot(df_large_sorted['LaunchDate'], df_large_sorted['Sales'], 'b-', alpha=0.6, label='Sales
ax7.set_xlabel('Launch Date')
ax7.set_ylabel('Price ($)', color='g')
ax8.set_ylabel('Sales', color='b')
ax7.set_title('Price and Sales Over Time')
plots_summary.append(('Dual Axis Plot - Price and Sales Over Time', fig7))

# Example 8 - Rolling Average of Sales
fig8, ax8 = plt.subplots()
```

```
ax8.plot(df_large_sorted['LaunchDate'], df_large_sorted['Sales'], label='Actual Sales', alpha=
ax8.plot(df_large_sorted['LaunchDate'], df_large_sorted['RollingSales'], label='Rolling Avg (2
ax8.set_title('Rolling Average of Sales')
ax8.set_xlabel('Launch Date')
ax8.set_ylabel('Sales')
ax8.legend()
ax8.grid(True)
plots_summary.append(('Line Plot - Rolling Average of Sales', fig8))

# Example 9 - Subplots with Mixed Chart Types
fig9, axs = plt.subplots(2, 2, figsize=(12, 10))
# Bar Chart - Model Count per Country
df_large['Country'].value_counts().plot(kind='bar', ax=axs[0, 0], color='lightgreen')
axs[0, 0].set_title('Model Count per Country')
# Scatter Plot - Price vs Total Revenue
axs[0, 1].scatter(df_large['Price'], df_large['TotalRevenue'], alpha=0.3, color='purple')
axs[0, 1].set_title('Price vs Total Revenue')
# Histogram - Sales Distribution
axs[1, 0].hist(df_large['Sales'], bins=20, color='orange')
axs[1, 0].set_title('Sales Distribution')
# Pie Chart - Country Distribution
df_large['Country'].value_counts().plot(kind='pie', autopct='%1.1f%%', ax=axs[1, 1])
axs[1, 1].set_ylabel('')
axs[1, 1].set_title('Country Distribution')
plt.tight_layout()
plots_summary.append(('Subplots - Mixed Chart Types', fig9))

# Displaying plots to the user
for title, fig in plots_summary:
    fig.suptitle(title)
    fig.tight_layout()
    fig.show()
```

Exercise

Lesson 19 — Hands-On Exercises with Matplotlib

Solutions (Code Examples)

```
import matplotlib.pyplot as plt

# 1
plt.plot(df['Model'], df['Sales'])
plt.title('Sales Over Models')
plt.xlabel('Model')
plt.ylabel('Sales')
plt.show()

# 2
```

```
plt.bar(df['Model'], df['Price'])
plt.title('Price per Model')
plt.show()

# 3
plt.barh(df['Model'], df['Sales'])
plt.title('Sales per Model (Horizontal)')
plt.show()

# 4
plt.scatter(df['Price'], df['Sales'])
plt.title('Price vs Sales')
plt.xlabel('Price')
plt.ylabel('Sales')
plt.show()

# 5
plt.hist(df['Rating'])
plt.title('Histogram of Ratings')
plt.show()

# 6
plt.boxplot(df['Price'])
plt.title('Boxplot of Price')
plt.show()

# 7
df['Type'].value_counts().plot.pie(autopct='%1.1f%%')
plt.title('Vehicle Type Distribution')
plt.ylabel('')
plt.show()

# 8
df.groupby('Type')['Price'].mean().plot.bar()
plt.title('Average Price per Type')
plt.show()

# 9
plt.plot(df['LaunchDate'], df['Sales'])
plt.title('Sales Over Launch Dates')
plt.show()

# 10
df['Country'].value_counts().plot.bar()
plt.title('Number of Models per Country')
plt.show()

# 11
```

```
plt.scatter(df['Price'], df['Sales'], c=pd.factorize(df['Type'])[0])
plt.title('Price vs Sales Colored by Type')
plt.show()

# 12
df.groupby(['Country', 'Type'])['Sales'].sum().unstack().plot(kind='bar')
plt.title('Sales by Country and Type')
plt.show()

# 13
df.groupby(['Country', 'Type'])['Sales'].sum().unstack().plot(kind='bar', stacked=True)
plt.title('Stacked Sales by Type per Country')
plt.show()

# 14
for t, group in df.groupby('Type'):
    plt.plot(group['LaunchDate'], group['Sales'], label=t)
plt.legend()
plt.title('Sales Trends by Type')
plt.show()

# 15
plt.scatter(df['Price'], df['Sales'], s=df['Rating']*20)
plt.title('Scatter with Rating as Marker Size')
plt.show()

# 16
df['Country'].value_counts().plot.pie(autopct='%1.1f%%')
plt.title('Country-wise Model Share')
plt.ylabel('')
plt.show()

# 17
plt.hist(df['Price'], bins=5)
plt.title('Histogram of Prices (5 Bins)')
plt.show()

# 18
df.boxplot(column='Rating', by='Type')
plt.title('Boxplot of Ratings by Type')
plt.suptitle('')
plt.show()

# 19
plt.scatter(df['Price'], df['Sales'], c=df['Rating'], cmap='viridis')
plt.colorbar(label='Rating')
plt.title('Sales vs Price Colored by Rating')
plt.show()
```

```
# 20
df.sort_values('LaunchDate', inplace=True)
plt.plot(df['LaunchDate'], df['Sales'].cumsum())
plt.title('Cumulative Sales Over Time')
plt.show()

# 21
bars = plt.bar(df['Model'], df['Sales'])
plt.title('Sales with Annotations')
for bar in bars:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, yval, int(yval), ha='center', va='bottom')
plt.show()

# 22
df['RollingSales'] = df['Sales'].rolling(3).mean()
plt.plot(df['LaunchDate'], df['Sales'], label='Actual')
plt.plot(df['LaunchDate'], df['RollingSales'], label='Rolling Avg')
plt.legend()
plt.title('Sales with Rolling Average')
plt.show()

# 23
fig, ax1 = plt.subplots()
ax2 = ax1.twinx()
ax1.plot(df['LaunchDate'], df['Price'], 'g-')
ax2.plot(df['LaunchDate'], df['Sales'], 'b-')
ax1.set_xlabel('Launch Date')
ax1.set_ylabel('Price', color='g')
ax2.set_ylabel('Sales', color='b')
plt.title('Dual Axis Plot')
plt.show()

# 24
import seaborn as sns
sns.heatmap(df[['Price', 'Sales', 'Rating']].corr(), annot=True, cmap='coolwarm')
plt.title('Correlation Heatmap')
plt.show()

# 25
fig, axs = plt.subplots(2, 2, figsize=(10, 10))
axs[0, 0].bar(df['Model'], df['Sales'])
axs[0, 0].set_title('Sales per Model')
axs[0, 1].scatter(df['Price'], df['Sales'])
axs[0, 1].set_title('Price vs Sales')
axs[1, 0].hist(df['Rating'])
axs[1, 0].set_title('Histogram of Ratings')
```

```
df['Type'].value_counts().plot.pie(ax=axs[1, 1], autopct='%1.1f%%')
axs[1, 1].set_title('Vehicle Type Distribution')
plt.tight_layout()
plt.show()
```

Exercise

Lesson 19 — Hands-On Exercises with Matplotlib

Data:

```
import numpy as np
import pandas as pd

# Define the number of synthetic rows to generate
num_rows = 1000

# Possible values to sample from
models = ['Alpha', 'Beta', 'Gamma', 'Delta', 'Epsilon', 'Zeta', 'Eta', 'Theta', 'Iota', 'Kappa
          'Nu', 'Xi', 'Omicron', 'Pi', 'Rho', 'Sigma', 'Tau', 'Upsilon', 'Phi', 'Chi', 'Psi',
types = ['Hatchback', 'Sedan', 'SUV', 'Truck']
countries = ['Japan', 'Germany', 'USA']

# Random data generation
np.random.seed(42)
extended_data = {
    'Model': np.random.choice(models, num_rows),
    'Price': np.random.randint(15000, 40000, num_rows),
    'Sales': np.random.randint(100, 1000, num_rows),
    'Rating': np.round(np.random.uniform(2.5, 5.0, num_rows), 1),
    'Type': np.random.choice(types, num_rows),
    'Country': np.random.choice(countries, num_rows),
    'LaunchDate': pd.to_datetime(np.random.choice(pd.date_range('2021-01-01', '2025-01-01'), nu
}

df_large = pd.DataFrame(extended_data)
df_large.head()
```

Assignments with Titles & Purposes

1. **Line Plot of Sales Over Models** Plot a simple line chart to visualize how sales vary across different vehicle models.
2. **Bar Chart of Price per Model** Create a vertical bar chart showing the price of each vehicle model.
3. **Horizontal Bar Chart of Sales per Model** Use a horizontal bar chart to represent sales data per model — ideal for longer model names.

4. **Scatter Plot of Price vs Sales** Visualize the relationship between vehicle price and sales with a scatter plot.
5. **Histogram of Ratings** Display the distribution of customer ratings using a histogram.
6. **Boxplot of Prices** Plot a boxplot to detect outliers and visualize the spread of vehicle prices.
7. **Pie Chart of Vehicle Type Distribution** Show the proportion of different vehicle types in the dataset with a pie chart.
8. **Bar Chart of Average Price per Vehicle Type** Calculate the average price for each type of vehicle and display it in a bar chart.
9. **Line Chart of Sales Over Launch Dates** Visualize sales over time using a line chart with launch dates on the x-axis.
10. **Bar Chart of Model Count per Country** Plot the number of models produced per country using a bar chart.
11. **Scatter Plot with Color-Coded Vehicle Types** Add a color-coded category to a scatter plot to distinguish vehicle types visually.
12. **Grouped Bar Chart of Sales by Country and Type** Use a grouped bar chart to compare sales by country and vehicle type.
13. **Stacked Bar Chart of Sales by Type within Each Country** Represent cumulative sales per country with a stacked bar chart showing vehicle types.
14. **Multiple Line Charts of Sales Trends per Vehicle Type** Plot individual sales trends over time for each vehicle type on the same chart.
15. **Create "Total Revenue" Column and Scatter Plot with Marker Size** Add a column calculating total revenue (Price $\times$ Sales) and plot a scatter chart with marker sizes representing revenue.
16. **Pie Chart of Market Share by Country** Aggregate data to display each country's share of total models in a pie chart.
17. **Histogram of Prices with Custom Bin Size** Adjust the number of bins in a histogram to control grouping of price values.
18. **Boxplot of Ratings Grouped by Vehicle Type** Compare distributions of ratings for each vehicle type using grouped boxplots.
19. **Create "Profit Margin" Column and Scatter Plot with Color Encoding** After creating a new column for profit margin (e.g., Revenue divided by Price), visualize it on a scatter plot using color to encode margin.
20. **Cumulative Sales Over Time Line Plot** Calculate cumulative sales and plot them over time to analyze growth trends.
21. **Bar Chart with Data Labels (Annotations)** Enhance a bar chart by adding labels directly on top of each bar for better readability.
22. **Rolling Average of Sales with Line Chart** Compute a rolling average of sales (e.g., over 3 models) and plot both the original and smoothed data.

23. **Dual Axis Line Plot of Price and Sales Over Time** Plot price and sales on the same chart using two y-axes for clear comparison.
24. **Time Series of Total Revenue with Cumulative Sum** Plot a cumulative sum of total revenue over time to visualize income growth.
25. **Subplots with Mixed Chart Types in One Figure** Practice using subplots to display multiple chart types (e.g., bar chart, scatter plot, histogram, pie chart) in a single figure.

Lesson 20

1. Handling Exceptions In Python

1. Handling Exceptions in Python

Example: Division with Exception Handling

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
    print("Result:", result)
except ValueError:
    print("Invalid input. Please enter an integer.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
finally:
    print("Operation complete.")
```

Extended Explanation

Component	Purpose
`try:`	Block of code where you anticipate an error might occur.
`int(input(...))`	Converts the input string to an integer. May raise `ValueError`.
`10 / num`	Risky operation that may raise `ZeroDivisionError` if `num == 0`.
`except ValueError`	Catches cases where user input is not a valid integer (e.g., "abc").
`except ZeroDivisionError`	Handles mathematical error of dividing by zero.
`finally:`	Runs regardless of whether an error occurred (commonly used for cleanup).

Common Python Errors & Exceptions

---

1. `ZeroDivisionError`

Raised when you attempt to divide by zero.

Example:

```
result = 10 / 0
```

Handling:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
```

Use Case: Preventing crashes in math operations, calculators, billing systems.

---

2. `ValueError`

Occurs when a function receives the correct type of argument but the **wrong value**.

Example:

```
age = int("twenty")  # Can't convert string to integer
```

Handling:

```
try:
    age = int(input("Enter your age: "))
except ValueError:
    print("Please enter a valid number.")
```

Use Case: Validating user input from forms or CLI.

---

3. `TypeError`

Raised when an operation or function is applied to an object of **incompatible type**.

Example:

```
result = "5" + 3  # Can't add string and integer
```

Handling:

```
try:
    result = "5" + 3
except TypeError:
    print("Type mismatch: You cannot add string and integer.")
```

Use Case: Useful in APIs or libraries where types must be strictly controlled.

---

4. `IndexError`

Occurs when trying to access an **invalid index** in a list or tuple.

Example:

```
items = [1, 2, 3]
print(items[5])  # Index out of range
```

Handling:

```
try:
    print(items[5])
except IndexError:
    print("Index is out of bounds.")
```

Use Case: Accessing list data from user input or dynamic loops in data processing.

---

5. `KeyError`

Happens when trying to access a **non-existent key** in a dictionary.

Example:

```
data = {"name": "Alice"}
print(data["age"])  # Key "age" not found
```

Handling:

```
try:
    print(data["age"])
except KeyError:
    print("Key not found in dictionary.")
```

Use Case: Data validation in configurations, JSON APIs, or form parsing.

---

6. `FileNotFoundError`

Occurs when trying to **open a file** that doesn't exist.

Example:

```
file = open("nonexistent.txt", "r")
```

Handling:

```
try:
    file = open("nonexistent.txt", "r")
except FileNotFoundError:
    print("The file does not exist.")
```

Use Case: File loading systems, user-uploaded data, or config-based apps.

---

7. `AttributeError`

Raised when you try to access a **method or attribute that doesn't exist** for an object.

Example:

```
num = 10
num.append(5)  # 'int' object has no attribute 'append'
```

Handling:

```
try:
    num.append(5)
except AttributeError:
    print("Invalid attribute access.")
```

Use Case: Dynamic object handling, reflection, or plugin-based architectures.

---

8. `ImportError` / `ModuleNotFoundError`

Occurs when a Python module **can't be imported** (e.g., it's missing or not installed).

Example:

```
import nonexistingmodule
```

Handling:

```
try:
    import nonexistingmodule
except ImportError:
    print("Module not found.")
```

Use Case: Plugin systems, conditional module loading, app startup checks.

Best Practices for Handling Exceptions

- **Use specific exceptions**: Avoid using `except:` alone.
- **Log or report errors**: Use `logging` for larger applications.
- **Don't suppress errors silently**: Always give feedback or handle the issue.
- **Use `finally`** for cleanup: e.g., closing files or database connections.

Alternative Scenario with Multiple Exception Types

```
try:
    num1 = input("Enter a number: ")
    num2 = input("Enter another number: ")
    result = int(num1) / int(num2)
    print("The result is:", result)
except ValueError as e:
    print("Conversion error:", e)
except ZeroDivisionError as e:
    print("Math error:", e)
except Exception as e:
    print("Unexpected error:", e)
finally:
    print("Done with calculation.")
```

What's New:

- Handles any unexpected error using `except Exception as e`.
- Captures the exact error message for debugging.
- More general-purpose and safer in real-world apps.

Custom Error Messages

You can also raise custom errors or handle specific situations based on logic:

```
try:
    age = int(input("Enter your age: "))
    if age < 0:
        raise ValueError("Age cannot be negative.")
    print("You entered:", age)
except ValueError as e:
    print("Error:", e)
finally:
    print("Input process completed.")
```

Practical Use Cases

Scenario	Use of Exception Handling
Online Calculator Tool	Prevents division by zero or invalid input from crashing the app.
Form Submission Validation	Catches and reports incorrect data types (e.g., text in a number field).
Data Parsing in Files	Safely reads numeric values from text or CSV files that may contain missing or corrupted data.
Banking App Input Fields	Ensures numeric fields (e.g., amount, account number) are correctly formatted.
APIs or Web Services	Gracefully handles input/output errors or unexpected responses during runtime.

Best Practices for Error Handling

- **Be specific**: Catch specific exceptions instead of a generic `except`.
- **Use `finally`**: Always include cleanup logic (like closing files, logging, etc.).
- **Don't ignore errors**: At the very least, log them.
- **Avoid overusing `try`**: Handle only the parts of code that may fail, not entire blocks unnecessarily.

Ii.Errors And Exceptions

II. Errors and exceptions

Sample Dataset (for demo programs)

Use this list of diverse values in examples:

```
sample = ["42", "0", "hello", -1, None, 3.14, [], {}, 7]
```

This dataset contains strings valid for integer conversion, zero, invalid strings, negative values, floats, empty lists/dicts, and a good integer.

---

1. How `try…except…else…finally` Works

According to the Python docs:

- **`try` clause** executes code that may raise an exception.
- If **no exception** occurs, any `except` blocks are skipped, and control moves to `else` (if present), then `finally`.
- If an exception **occurs**, the rest of the `try` block is skipped; execution jumps to the **first matching `except` clause**. If none matches, the exception propagates out.
- The **`else` clause** runs only when the `try` succeeds without exceptions.
- The **`finally` clause** always executes, whether or not an exception occurred. Even if `except` or `else` raises an error, `finally` still runs before propagation.

- ---

2. Demo Programs Covering Exception Types

A: Integer Conversion and Division Loop

```
import sys

for item in sample:
    try:
        print(f"Testing: {item} → int:", end=" ")
        n = int(item)
        print(n, "→ divide 100 by it:", end=" ")
        result = 100 / n
    except ValueError as e:
        print("ValueError:", e)
    except ZeroDivisionError as e:
        print("ZeroDivisionError:", e)
    else:
        print("Result =", result)
    finally:
        print("- End of processing for this item\n")
```

What it shows:

- Convert items like `"hello"` → raises `ValueError`
- Divide by `"0"` → raises `ZeroDivisionError`
- For valid int and non-zero → runs `else`
- Always runs `finally`

B: Handling `TypeError`, `KeyError`, `IndexError`

```
# TypeError demo
a = [1, 2, 3]
try:
    print("Adding int and list:", 1 + a)
except TypeError as e:
    print("TypeError:", e)

# IndexError demo
try:
    print("Accessing index:", a[10])
except IndexError as e:
    print("IndexError:", e)

# KeyError demo
d = {"x": 1}
try:
```

```
    print("Access key 'y':", d["y"])
except KeyError as e:
    print("KeyError:", e)
finally:
    print("Completed lookup demos\n")
```

C: FileNotFoundError Clean-up with `finally`

```
try:
    f = open("missing_file.txt", "r")
    data = f.read()
except FileNotFoundError as e:
    print("FileNotFoundError:", e)
else:
    print("File length:", len(data))
finally:
    print("Cleaning up: closing file if needed")
    try:
        f.close()
    except NameError:
        print("File was never opened")
```

D: Custom Raising and Re-Raising Exceptions

```
def validate_positive(n):
    if n < 0:
        raise ValueError(f"Negative value not allowed: {n}")
    return n

try:
    x = validate_positive(-5)
except ValueError as e:
    print("Caught custom ValueError:", e)
    raise  # Re-raises the same exception
finally:
    print("Done validate_positive demo")
```

---

3. Summary Table

Clause	Runs When?	Purpose
`try`	Always	Put code that might raise exceptions
`except`	Only if a matching exception occurs	Handle specific errors (e.g., `ValueError`, `ZeroDivisionError`)

Clause	Runs When?	Purpose
`else`	Only if `try` succeeds with no exception	Code to run when everything went fine
`finally`	Always, before block ends	Cleanup such as closing files or resources

Use specific exception types where possible (e.g. `except ValueError`), not generic `except:`. Use `else` to separate success logic, and `finally` for cleanup ([Python documentation][1], [Stack Overflow][4], [Medium][5], [Wikibooks][3], [Python documentation][6], [Python documentation][7]).

---

4. Recap of Covered Built-in Exceptions

From the tutorials and docs:

- **`ValueError`**: right type, wrong value (e.g. `int("hello")`)
- **`ZeroDivisionError`**: dividing by zero
- **`TypeError`**: unsupported operand type(s)
- **`IndexError`**, **`KeyError`**: invalid list/dict access
- **`FileNotFoundError`**: missing file on open
- **`Exception` / re-raise**: use generic catch with caution; good for logging then re-raising ([deepnote.com][8])

Putting It All Together: Full Script Example

```
import sys

sample = ["42", "0", "hello", -1, None, 3.14, [], {}, 7]

def validate_positive(n):
    if n < 0:
        raise ValueError(f"Negative value not allowed: {n}")
    return n

for item in sample:
    try:
        print("Item:", item)
        n = int(item)
        validate_positive(n)
        result = 100 / n
    except ValueError as e:
        print("ValueError:", e)
    except ZeroDivisionError as e:
        print("ZeroDivisionError:", e)
    except Exception as e:
        print("Unexpected exception:", type(e).__name__, "-", e)
```

```
    else:
        print("OK → result =", result)
    finally:
        print("Finished processing this item.")
        print("-" * 40)
```

Iii. Exceptions

III. Exceptions

What is an Exception?

An **exception** is an error that **interrupts normal program flow**. Instead of crashing the program, Python allows you to handle exceptions using `try`, `except`, and related keywords.

Common Built-in Exceptions

Exception	Description
`ZeroDivisionError`	Raised when dividing by zero
`ValueError`	Raised when a function gets an argument of correct type but invalid value
`TypeError`	Raised when an operation or function is applied to an object of inappropriate type
`IndexError`	Raised when a sequence index is out of range
`KeyError`	Raised when a dictionary key is not found
`FileNotFoundError`	Raised when trying to open a file that doesn't exist
`ImportError`	Raised when an import fails
`AttributeError`	Raised when an object does not have a requested attribute

---

Demo 1: Catching Multiple Exception Types

```
data = ["10", "0", "abc", None]

for item in data:
    try:
        print(f"Processing: {item}")
        number = int(item)
        result = 100 / number
    except ValueError:
        print("ValueError: Cannot convert to integer.")
    except ZeroDivisionError:
        print("ZeroDivisionError: Cannot divide by zero.")
    except Exception as e:
        print(f"Unhandled Exception: {type(e).__name__} - {e}")
```

```
    finally:
        print("Done.\n")
```

Demo 2: Catching Unknown Exceptions

```
try:
    x = 5
    y = "10"
    z = x + y  # Will raise TypeError
except Exception as e:
    print(f"Exception caught: {type(e).__name__} - {e}")
```

** Tip:** Using `except Exception as e` is helpful for debugging unknown issues, but not recommended for production error handling unless logging and re-raising.

Demo 3: Using `else` and `finally`

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except (ValueError, ZeroDivisionError) as e:
    print("Caught an error:", e)
else:
    print("Everything worked! Result is:", result)
finally:
    print("This runs no matter what.")
```

Best Practices for Exceptions

- Catch **only expected exceptions**.
- Use **specific exception types** instead of a generic `except:`.
- Always use `finally` when a resource needs to be cleaned up (e.g., file closing).
- `raise` can be used to throw an exception manually.
- You can also define your own **custom exception classes** if needed.

Demo 4: Custom Exception

```
class NegativeNumberError(Exception):
    pass

def check_positive(n):
    if n < 0:
        raise NegativeNumberError("Negative number not allowed.")
```

```
    return n

try:
    check_positive(-5)
except NegativeNumberError as e:
    print("Custom exception caught:", e)
```

Iv. Handling And Raising Exceptions

IV. Handling and raising exceptions

What Is Exception Handling?

Exception handling allows you to **gracefully respond** to errors without crashing your program. This is done using:

- `try`: code that might raise an exception
- `except`: code to handle the exception
- `else`: code to run if no exceptions were raised
- `finally`: code that runs no matter what (for cleanup)

Basic Structure

```
try:
    # risky code
except SomeException:
    # handle error
else:
    # run if no error
finally:
    # always runs
```

Demo 1: Handling a Specific Exception

```
try:
    number = int(input("Enter a number: "))
    print("100 divided by your number is:", 100 / number)
except ValueError:
    print("Oops! That was not a valid number.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

Explanation:

- **`ValueError`**: occurs if input is non-numeric (e.g., “abc”)
- **`ZeroDivisionError`**: occurs if user enters `0`

Demo 2: Handling Multiple Exceptions in One Line

```
try:
    num = int(input("Enter a number: "))
    print("Reciprocal is", 1 / num)
except (ValueError, ZeroDivisionError) as e:
    print("Handled error:", e)
```

Explanation:

You can group multiple exceptions using parentheses.

Demo 3: Using `else` and `finally`

```
try:
    value = int(input("Enter an integer: "))
except ValueError:
    print("That's not an integer.")
else:
    print("Success! You entered:", value)
finally:
    print("Finished input processing.")
```

Explanation:

- `else`: runs only if no exception occurs.
- `finally`: runs **regardless** of whether an error was raised.

Demo 4: Handling Unknown Exceptions

```
try:
    risky = [1, 2, 3][5]  # IndexError
except Exception as e:
    print("An unexpected error occurred:", type(e).__name__, "-", e)
```

Tip:

- Using `except Exception` is good for logging or debugging.
- Avoid using it in place of specific exception handling.

Re-raising Exceptions

Sometimes you want to handle an error but still let it bubble up:

```
try:
    raise ValueError("Something went wrong.")
```

```
except ValueError as e:
    print("Logging error:", e)
    raise  # Re-raise the same exception
```

Best Practices

Practice	Why It Matters
Use specific exception types	Easier to debug, avoids catching unexpected errors
Use `finally` for cleanup	Guarantees resources (like files) are released
Log or raise meaningful exceptions	Helps with debugging and production error reporting
Avoid bare `except:` clauses	They hide real bugs and make issues harder to trace

---

Challenge Demo: Full Exception Handling Flow

```
def get_number():
    try:
        num = int(input("Enter an integer: "))
        if num == 0:
            raise ZeroDivisionError("Manual raise: zero not allowed.")
        print("Result is", 100 / num)
    except ValueError:
        print("Please enter a numeric value.")
    except ZeroDivisionError as e:
        print("Math error:", e)
    else:
        print("Everything worked fine!")
    finally:
        print("Cleanup complete.\n")

# Run the function
get_number()
```

Keyboardinterrupt

`KeyboardInterrupt` – What Is It?

`KeyboardInterrupt` is a built-in exception in Python that is raised **when the user interrupts program execution**, typically by pressing **Ctrl+C** (or **Cmd+C** on macOS) in the terminal.

Behavior

- Python raises `KeyboardInterrupt` when it receives the **SIGINT** signal (interrupt signal).
- It is a subclass of `BaseException`, **not** `Exception`, so it behaves differently when you use `except Exception`.

Demo: Catching `KeyboardInterrupt`

```
try:
    print("Press Ctrl+C to interrupt...")
    while True:
        pass  # Infinite loop
except KeyboardInterrupt:
    print("\nKeyboardInterrupt caught. Exiting gracefully.")
finally:
    print("Clean-up complete.")
```

Output if you press Ctrl+C during the loop:

```
Press Ctrl+C to interrupt...
^C
KeyboardInterrupt caught. Exiting gracefully.
Clean-up complete.
```

Special Note:

If you use a **generic `except Exception`**, it will **not catch `KeyboardInterrupt`**:

```
try:
    while True:
        pass
except Exception:
    print("This will NOT catch Ctrl+C")
```

This won't catch the interrupt, because `KeyboardInterrupt` inherits from `BaseException`, not `Exception`.

If You Want to Catch *All* Errors, Including Interrupts:

```
try:
    while True:
        pass
except BaseException as e:
    print("Caught:", type(e).__name__)
```

This will catch **everything**, including `SystemExit`, `KeyboardInterrupt`, etc. Use with caution—only if you really mean to handle all termination signals.

Should You Raise It Yourself?

Technically, yes—you can raise it like any exception:

```
raise KeyboardInterrupt("Simulating Ctrl+C")
```

But it's **unusual and discouraged** to manually raise `KeyboardInterrupt`. It's meant to reflect an actual user-initiated interrupt.

Use it only if you're simulating signals or testing interrupt handlers.

V. Clean-Up Actions

V. Clean-up actions

What is a Clean-up Action?

A **clean-up action** ensures that certain code (like closing files or releasing resources) **always runs**, even if an exception is raised.

Use Case: Resource Management

Resources like files, sockets, or database connections **must be released** even if an error occurs. Failing to do this can lead to:

- Resource leaks
- File locks
- Incomplete transactions

Methods for Clean-up:

1. Using `finally` Clause

The `finally` block **always executes**, whether an exception was raised or not.

Demo 1: File Handling with `finally`

```
try:
    file = open("example.txt", "r")
    # Simulate error or normal read
    content = file.read()
    print(content)
except FileNotFoundError:
    print("File not found.")
finally:
    print("Closing file...")
    try:
        file.close()
    except NameError:
        print("File was never opened.")
```

Explanation:

- `file.close()` is guaranteed to run.
- Prevents leaving a file open even when an error is raised.

---

2. Using `with` Statement (Context Manager)

The `with` statement is a cleaner and safer way to manage resources. It **automatically handles clean-up** by calling special methods like `__enter__()` and `__exit__()`.

Demo 2: File Handling with `with`

```
try:
    with open("example.txt", "r") as f:
        content = f.read()
        print(content)
except FileNotFoundError:
    print("File not found.")
```

Explanation:

- `with` automatically closes the file after the block.
- No need for `finally`.

Demo 3: `finally` Always Executes

```
try:
    print("Starting calculation...")
    result = 10 / 0
except ZeroDivisionError:
    print("Caught divide by zero.")
finally:
    print("Clean-up: Shutting down safely.")
```

Output:

```
Starting calculation...
Caught divide by zero.
Clean-up: Shutting down safely.
```

Summary Table

Approach	Description	Example Use Cases
`finally` block	Always runs. Useful for manual cleanup	File closing, DB disconnect, logging
`with` statement	Automatically handles clean-up via context managers	File I/O, network sockets, temporary files

---

Demo 4: Cleaning Up a Network Connection (Simulated)

```
class DummyConnection:
    def __enter__(self):
        print("Opening connection...")
        return self
    def __exit__(self, exc_type, exc_val, exc_tb):
        print("Closing connection...")

with DummyConnection() as conn:
    print("Using connection...")
    raise RuntimeError("Connection error!")
```

Output:

```
Opening connection...
Using connection...
Closing connection...
```

Even though an exception is raised, the clean-up runs!

Best Practices

- **Use `with` when possible** – it's concise, clean, and less error-prone.
- **Use `finally`** for manual or conditional resource release.
- **Never rely on garbage collection** to clean up important resources.

Vi. Modules

VI. Modules

What is a Module?

A **module** is a file containing Python code — variables, functions, classes — that you can **import and reuse** in other Python programs.

- File name: `mymodule.py`
- Allows code to be organized logically and reused across multiple files.

---

1. Creating a Module

Example: `calculator.py`

```python
# calculator.py
def add(a, b):
    return a + b

def multiply(a, b):
    return a * b

pi = 3.14159
```

---

2. Importing Modules

A. Basic Import

```python
import calculator

print(calculator.add(2, 3))        # 5
print(calculator.pi)               # 3.14159
```

B. Import with Alias

```python
import calculator as calc

print(calc.multiply(3, 4))         # 12
```

C. Import Specific Functions or Variables

```python
from calculator import add, pi

print(add(5, 6))                   # 11
print(pi)                          # 3.14159
```

D. Import All Symbols (*not recommended*)

```python
from calculator import *

print(multiply(2, 8))              # 16
```

** Warning:** This can **clutter your namespace** and lead to conflicts.

---

3. Module Search Path

Python searches for modules in these locations:

1. The current directory
2. Built-in modules (like `math`, `random`, etc.)

3. Directories listed in `sys.path`

You can inspect the module search path:

```
import sys
print(sys.path)
```

---

4. The `__name__ == "__main__"` Trick

Used to make a Python file work both as a **script** and a **module**.

Example: `greeter.py`

```
def greet(name):
    print(f"Hello, {name}!")

if __name__ == "__main__":
    greet("Alice")  # Runs only when executed directly
```

When you run:

```
python greeter.py
```

Output: `Hello, Alice!`

But if you import it:

```
import greeter
# greet() doesn't run automatically
```

---

5. Standard Modules

Python comes with a large library of **built-in modules**:

Module	Purpose	Example Usage
`math`	Mathematical functions	`math.sqrt(25)`
`random`	Random number generation	`random.randint(1, 10)`
`os`	File system, environment variables	`os.getcwd()`
`sys`	Interpreter info	`sys.version`, `sys.path`
`datetime`	Dates and times	`datetime.datetime.now()`

---

6. Packages (Brief Overview)

A **package** is a folder containing modules and a special `__init__.py` file (can be empty) to mark it as importable.

Folder Structure:

```
myapp/

   __init__.py
   math_utils.py
   string_utils.py
```

Then import like this:

```
from myapp import math_utils
```

Practical Demo: Using Standard Modules

Using `math` and `random`

```
import math
import random

angle = random.uniform(0, math.pi)
print(f"sin({angle:.2f}) = {math.sin(angle):.4f}")
```

Using `datetime`

```
from datetime import datetime

now = datetime.now()
print("Current time:", now.strftime("%Y-%m-%d %H:%M:%S"))
```

Summary Table

Action	Syntax
Import entire module	`import module`
Import with alias	`import module as alias`
Import specific items	`from module import item`
Import everything (not ideal)	`from module import *`
Check if script is main	`if __name__ == "__main__":`

Best Practices

- Use specific imports for clarity (`from module import item`)
- Use `as` to shorten long module names
- Avoid wildcard imports (`from module import *`)
- Use `__name__ == "__main__"` to separate testing from import behavior

Python Modules and Packages, Requisites, and Environments

---

1. What Are Python Modules and Packages?

- A **module** is a `.py` file containing Python code (functions, classes, variables).
- A **package** is a **directory** with an `__init__.py` file that can contain multiple modules or sub-packages.

Example Folder Structure:

```
my_project/

   calculator.py
   greeter.py
   main.py
   utils/
       __init__.py
       text_tools.py
```

You can import like:

```
from utils.text_tools import capitalize_words
```

---

2. What Are Requisites?

Requisites are **external packages** your code depends on (e.g., `numpy`, `requests`, `pandas`). These are not built into Python and must be installed using **pip**.

You track them using a `requirements.txt` file.

Example: `requirements.txt`

```
requests==2.31.0
pandas>=1.5
numpy
```

Code Using Requisites (`http_demo.py`)

```
import requests

def fetch_python_org():
    response = requests.get("https://www.python.org")
    if response.status_code == 200:
        print("Python.org fetched successfully!")
    else:
```

```
        print("Failed to reach Python.org")

fetch_python_org()
```

To run this:

1. Install dependencies:

   ```
   pip install -r requirements.txt
   ```

2. Run the script:

   ```
   python http_demo.py
   ```

How to Generate `requirements.txt`

From a working environment:

```
pip freeze > requirements.txt
```

This captures the current package versions you're using.

Example: Reading and Installing

Reading `requirements.txt`:

```
cat requirements.txt
```

Installing everything:

```
pip install -r requirements.txt
```

Virtual Environments (Isolated Python Spaces)

A **virtual environment** is a self-contained directory with its own Python binaries and packages. It helps you:

- Avoid version conflicts
- Reproduce environments across machines
- Keep project dependencies isolated

Creating a Virtual Environment

```
python -m venv venv
```

Activate it:

- **Windows**:

```
venv\Scripts\activate
```

- **Mac/Linux**:

```
source venv/bin/activate
```

After Activation:

```
pip install -r requirements.txt
```

Then run your program safely in an isolated space!

Recap Summary

Tool	Purpose	Command
`requirements.txt`	Track package dependencies	`pip freeze > requirements.txt`
`pip`	Install packages	`pip install package_name`
`venv`	Create isolated Python environments	`python -m venv venv`
`__init__.py`	Mark a directory as a Python package	Empty or with init logic
`import`	Use functions from modules	`from module import name`

Best Practices

- Always use a **virtual environment** per project.
- Use **specific versions** in `requirements.txt` for reproducibility.
- Keep related functionality in **separate modules** or packages.

Vii. Hands-On

VI. Hands-on

1. Create a folder structure for hands-on examples (simulated here in code)
2. Simulate the creation of multiple Python files as string examples

calculator.py (Module)

calculator_code:

```
def add(a, b):
    return a + b

def multiply(a, b):
    return a * b

pi = 3.14159
```

greeter.py (Script/Module with name check)

greeter_code:

```
def greet(name):
    print(f"Hello, {name}!")

if __name__ == "__main__":
    greet("Alice")
```

hands_on_demo.py (Main script using modules and exception handling)

hands_on_demo_code:

```
import calculator
import greeter
import math
import random
from datetime import datetime

def main():
    print("=== Calculator Demo ===")
    print("Add 5 + 3:", calculator.add(5, 3))
    print("Multiply 4 * 6:", calculator.multiply(4, 6))
    print("Value of pi:", calculator.pi)

    print("\\n=== Random and Math Demo ===")
    num = random.randint(1, 100)
    print("Random number:", num)
    print("Square root:", math.sqrt(num))

    print("\\n=== DateTime Demo ===")
    now = datetime.now()
    print("Current time:", now.strftime("%Y-%m-%d %H:%M:%S"))

    print("\\n=== Exception Handling Demo ===")
    sample = ["42", "0", "abc", None]
    for item in sample:
        try:
            print(f"Processing: {item}")
            n = int(item)
            print("Reciprocal:", 1 / n)
        except ValueError:
            print("ValueError: Cannot convert to int.")
        except ZeroDivisionError:
            print("ZeroDivisionError: Cannot divide by zero.")
        except Exception as e:
            print("Unexpected error:", type(e).__name__, "-", e)
        finally:
```

```
            print("Done.\\n")

if __name__ == "__main__":
    main()
```

Main file

```
import pandas as pd
import ace_tools as tools


examples_df = pd.DataFrame([
    {"File": "calculator.py", "Purpose": "Module with math functions", "Code": calculator_code
    {"File": "greeter.py", "Purpose": "Greets user; demonstrates __name__ check", "Code": greet
    {"File": "hands_on_demo.py", "Purpose": "Main hands-on script for demo", "Code": hands_on_c
])

tools.display_dataframe_to_user(name="Hands-On Python Modules and Exception Examples", datafra
```

Content

LESSON 20: ERROR HANDLING AND MODULES

- Handling exceptions in Python using `try`, `except`, `finally`
- Importing and using Python modules (`math`, `random`, etc.)
- Working with variables and objects
- Hands-on exercises

---

1. Handling Exceptions in Python

Example: Division with Exception Handling

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
    print("Result:", result)
except ValueError:
    print("Invalid input. Please enter an integer.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
finally:
    print("Operation complete.")
```

Explanation:

- `try`: runs code that may raise an error.
- `except ValueError`: handles non-integer inputs.
- `except ZeroDivisionError`: handles divide-by-zero errors.

- `finally`: always runs, whether there's an error or not.

Practical Use Case: User input validation in a calculator or online form to prevent crashes and give helpful feedback.

---

2. Importing and Using Modules

Example: Using `math` and `random` Modules

```
import math
import random

num = random.randint(1, 100)
square_root = math.sqrt(num)

print(f"The square root of {num} is {square_root:.2f}")
```

Explanation:

- `random.randint(1, 100)`: picks a random number between 1 and 100.
- `math.sqrt(num)`: calculates the square root of the number.

Practical Use Case: Games, simulations, statistical tools, or quiz apps that need randomness and math operations.

---

3. Working with Variables and Objects

Example: Object-Oriented Error Handling

```
class Calculator:
    def __init__(self, value):
        self.value = value

    def inverse(self):
        try:
            return 1 / self.value
        except ZeroDivisionError:
            return "Cannot take inverse of zero."

calc = Calculator(0)
print(calc.inverse())
```

Explanation:

- Shows working with object properties (`self.value`).
- `Calculator` is a custom object with methods that include error handling.

Practical Use Case: Encapsulate logic in classes (e.g., scientific calculator, finance apps) with robust error handling.

---

3. Logging in Python

Hands-On Practice Exercise

```
# Ask the user for a number and compute its square root
import math

try:
    user_input = input("Enter a number to find the square root: ")
    number = float(user_input)
    if number < 0:
        raise ValueError("Cannot compute square root of a negative number.")
    result = math.sqrt(number)
    print(f"Square root of {number} is {result:.2f}")
except ValueError as ve:
    print("Error:", ve)
finally:
    print("Thank you for using the calculator.")
```

Explanation:

- Demonstrates full use of exception handling and module import.
- Validates user input and handles math domain errors.

Lesson 21

Lesson 21 Error Handling And Modules – Solutions

Lesson 21: Error Handling and Modules – Exercises

Solutions

Solution 1

```
try:
    num = float(input("Enter a number: "))
    print("Square:", num ** 2)
except ValueError:
    print("Invalid input! Please enter a number.")
```

```
 try:
```

```
    #check_age(25)
    check_age(-5)
except ValueError as e:
    print("Error:", e)
```

Solution 2

```
try:
    a = float(input("Enter first number: "))
    b = float(input("Enter second number: "))
    print("Result:", a / b)
except ValueError:
    print("Invalid input! Enter numbers only.")
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

Solution 3

```
try:
    f = open("sample.txt", "r")
    print(f.read())
except FileNotFoundError:
    print("File not found.")
finally:
    try:
        f.close()
    except:
        pass
```

Solution 4

```
try:
    num = int("abc")
except ValueError:
    print("Cannot convert string to integer.")
```

Solution 5

```
def check_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative")
    print("Valid age")

check_age(25)
# check_age(-5) # Uncomment to test
```

Solution 6

```
def add_numbers(a, b):
    if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
        raise TypeError("Both arguments must be numbers")
    return a + b
```

```
print(add_numbers(5, 3))
```

Solution 7

```
def check_positive(num):
    assert num > 0, "Number must be positive"
    print("Number is positive.")

# check_positive(-5) # Will raise AssertionError
check_positive(10)
```

Solution 8

```
import logging

logging.basicConfig(level=logging.ERROR)

try:
    a = 10 / 0
except ZeroDivisionError as e:
    logging.error("Error occurred: %s", e)
```

Solution 9

```
import logging

logging.basicConfig(filename="errors.log", level=logging.ERROR)

filename = input("Enter filename: ")
try:
    with open(filename, "r") as f:
        print(f.read())
except Exception as e:
    logging.error("Error opening file: %s", e)
```

Solution 10

```
import logging

logging.basicConfig(filename="conversion.log", level=logging.ERROR)

data = ["1", "2", "x", "4"]
for item in data:
    try:
        num = int(item)
    except ValueError as e:
        logging.error("Conversion error for '%s': %s", item, e)
```

Solution 11

```
import math
```

```
num = int(input("Enter a number: "))
print("Square root:", math.sqrt(num))
print("Factorial:", math.factorial(num))
```

Solution 12

```
import random

target = random.randint(1, 100)
guess = None
while guess != target:
    guess = int(input("Guess the number (1-100): "))
    if guess < target:
        print("Too low!")
    elif guess > target:
        print("Too high!")
print("Correct!")
```

Solution 13

```
from datetime import datetime, timedelta

now = datetime.now()
print("Now:", now)
print("7 days later:", now + timedelta(days=7))
```

Solution 14

```
import os

print("Current directory:", os.getcwd())
print("Files:", os.listdir())
```

Solution 15

```
import sys

print("Python version:", sys.version)
print("Script name:", sys.argv[0])
```

Solution 16

```
import math

functions = [f for f in dir(math) if not f.startswith("__")]
print(functions)
```

Solution 17

```
import importlib.metadata

print("Numpy version:", importlib.metadata.version("numpy"))
```

Solution 18 my_utils.py

```python
def greet(name):
    print(f"Hello, {name}!")
```

main.py

```python
import my_utils

my_utils.greet("Alice")
```

Solution 19

```python
import statistics

data = [1, 2, 2, 3, 4, 5, 5, 5]
print("Mean:", statistics.mean(data))
print("Median:", statistics.median(data))
print("Mode:", statistics.mode(data))
```

Solution 20

```python
import requests

response = requests.get("https://api.github.com")
print("Status code:", response.status_code)
```

Exercise

Lesson 21: Error Handling and Modules – Exercises

Part 1: Error Handling

Exercise 1 – Basic `try-except` Write a program that asks the user for a number and prints its square. Handle any errors if the input is not a valid number.

Exercise 2 – Multiple `except` Blocks Write a program that:

- Divides two numbers entered by the user
- Catches both `ValueError` (invalid input) and `ZeroDivisionError` separately

Exercise 3 – `try-except-finally` Write a program that opens a file and reads its content. Make sure the file is always closed, even if an error occurs.

Exercise 4 – Catching Specific Exceptions Write a program that tries to convert a string `"abc"` to an integer and catches the error.

Exercise 5 – Raising a `ValueError` Write a function `check_age(age)` that:

- Raises a `ValueError` if age is negative
- Prints "Valid age" otherwise

Exercise 6 – Raising a `TypeError` Write a function `add_numbers(a, b)` that:

- Raises a `TypeError` if either `a` or `b` is not a number
- Returns their sum otherwise

Exercise 7 – Using `assert` Write a function that asserts the given number is positive. Test it with a negative number.

Exercise 8 – Logging Errors (Console) Write a program that:

- Tries to divide two numbers
- Logs the error to the console using the `logging` module

Exercise 9 – Logging Errors (File) Write a program that:

- Reads a file name from the user
- Tries to open the file
- Logs any errors to a file `errors.log`

Exercise 10 – Combining `try-except` with Logging Write a program that:

- Tries to convert a list of strings to integers
- Logs all errors into a file `conversion.log`

Part 2: Working with Modules

Exercise 11 – Using the `math` Module Write a program that:

- Asks the user for a number
- Prints its square root and factorial using the `math` module

Exercise 12 – Using the `random` Module Write a program that:

- Generates a random number between 1 and 100
- Lets the user guess until they get it right

Exercise 13 – Using the `datetime` Module Write a program that:

- Prints the current date and time
- Prints the date 7 days from now

Exercise 14 – Using the `os` Module Write a program that:

- Prints the current working directory
- Lists all files in it

Exercise 15 – Using the `sys` Module Write a program that:

- Prints the Python version
- Prints the script name

Exercise 16 – Listing Functions in a Module Write a program that:

- Lists all available functions in the `math` module (excluding special methods)

Exercise 17 – Getting a Package Version Write a program that:

- Prints the version of the `numpy` package (if installed)

Exercise 18 – Creating and Importing a Custom Module Create a module `my_utils.py` with a function `greet(name)` that prints `"Hello, <name>!"`. Import and use it in another script.

Exercise 19 – Using the `statistics` Module Write a program that:

- Calculates the mean, median, and mode of a given list of numbers

Exercise 20 – Using a Third-Party Module Install the `requests` module and write a program that fetches data from `https://api.github.com` and prints the status code.

Error_Handling

Part 1

Assignment 1: Division Calculator

Task: Write a program that takes two numbers from the user and prints their division. Handle division by zero.

```
try:
    a = int(input("Enter numerator: "))
    b = int(input("Enter denominator: "))
    result = a / b
    print("Result:", result)
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
except ValueError:
    print("Error: Please enter valid integers.")
```

Assignment 2: File Reader

Task: Ask the user for a filename and print its contents. Handle the case where the file doesn't exist.

```
try:
    filename = input("Enter filename: ")
    with open(filename, 'r') as f:
        print(f.read())
except FileNotFoundError:
    print("Error: File not found.")
```

Assignment 3: List Indexing

Task: Create a list of 5 elements. Ask the user for an index and print the element. Handle index errors.

```
data = [10, 20, 30, 40, 50]

try:
    idx = int(input("Enter an index (0-4): "))
```

```
    print("Element:", data[idx])
except IndexError:
    print("Error: Index out of range.")
except ValueError:
    print("Error: Please enter a valid number.")
```

Assignment 4: Calculator with Multiple Exception Handling

Task: Build a calculator that takes two numbers and an operation (+, -, *, /). Handle input and operation errors.

```
try:
    a = float(input("Enter first number: "))
    b = float(input("Enter second number: "))
    op = input("Enter operation (+, -, *, /): ")

    if op == '+':
        print(a + b)
    elif op == '-':
        print(a - b)
    elif op == '*':
        print(a * b)
    elif op == '/':
        print(a / b)
    else:
        raise ValueError("Invalid operation.")
except ZeroDivisionError:
    print("Error: Division by zero.")
except ValueError as e:
    print("Error:", e)
```

Assignment 5: Using `else` and `finally`

Task: Perform a division with `try-except-else-finally` and explain the flow.

```
try:
    x = int(input("Enter numerator: "))
    y = int(input("Enter denominator: "))
    result = x / y
except ZeroDivisionError:
    print("Cannot divide by zero.")
else:
    print("Result:", result)
finally:
    print("Execution complete.")
```

Assignment 6: Raising an Exception

Task: Write a function that takes age as input. Raise a `ValueError` if age is negative.

```
def check_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative.")
    print("Age is valid.")

try:
    age = int(input("Enter your age: "))
    check_age(age)
except ValueError as e:
    print("Error:", e)
```

Assignment 7: Custom Exception

Task: Create a custom exception `PasswordTooShortError` and use it in a password checker.

```
class PasswordTooShortError(Exception):
    pass

def check_password(pwd):
    if len(pwd) < 6:
        raise PasswordTooShortError("Password must be at least 6 characters long.")
    print("Password accepted.")

try:
    pwd = input("Enter password: ")
    check_password(pwd)
except PasswordTooShortError as e:
    print("Error:", e)
```

Assignment 8: Nested Try Blocks

Task: Demonstrate use of nested `try` blocks.

```
try:
    a = int(input("Enter number: "))
    try:
        b = int(input("Enter divisor: "))
        print("Result:", a / b)
    except ZeroDivisionError:
        print("Inner Error: Division by zero.")
except ValueError:
    print("Outer Error: Invalid input.")
```

Assignment 9: Exception Logging

Task: Log exceptions to a file.

```
import logging

logging.basicConfig(filename='errors.log', level=logging.ERROR)

try:
    x = int(input("Enter number: "))
    y = int(input("Enter divisor: "))
    print("Result:", x / y)
except Exception as e:
    logging.error("Exception occurred", exc_info=True)
    print("An error occurred. Check errors.log for details.")
```

Assignment 10: Retry on Error

Task: Retry taking input until a valid integer is entered.

```
while True:
    try:
        num = int(input("Enter an integer: "))
        print("You entered:", num)
        break
    except ValueError:
        print("Invalid input. Try again.")
```

Part 2

Assignment 1: Using the `math` Module

Task: Import the `math` module and perform the following:

- Calculate the square root of 64
- Find the value of
- Calculate the sine of 90 degrees

```
import math

print("Square root of 64:", math.sqrt(64))
print("Value of pi:", math.pi)
print("Sine of 90 degrees:", math.sin(math.radians(90)))
```

Assignment 2: Working with `random` Module

Task: Use the `random` module to:

- Generate a random integer between 10 and 100
- Shuffle a list
- Pick a random element from a list

```
import random

numbers = [1, 2, 3, 4, 5]

print("Random integer:", random.randint(10, 100))
random.shuffle(numbers)
print("Shuffled list:", numbers)
print("Random choice:", random.choice(numbers))
```

Assignment 3: Using `datetime` Module

Task: Use the `datetime` module to:

- Get the current date and time
- Print only the current year
- Add 7 days to the current date

```
import datetime

now = datetime.datetime.now()
print("Current date and time:", now)
print("Current year:", now.year)
future = now + datetime.timedelta(days=7)
print("Date after 7 days:", future)
```

Assignment 4: Get Module Information with `dir()` and `help()`

Task: Use `dir()` and `help()` on the `math` module. Print the first 5 items returned by `dir()`.

```
import math

print("First 5 attributes of math module:", dir(math)[:5])
# Uncomment the next line to explore help in an interactive session
# help(math)
```

Assignment 5: Use the `os` Module to List Files

Task: Use the `os` module to:

- Print the current working directory
- List all files in the current directory

```
import os

print("Current working directory:", os.getcwd())
print("Files in directory:", os.listdir())
```

Assignment 6: Checking Module Versions

Task: Print the version of the `sys` module and check the version of Python being used.

```
import sys

print("Python version:", sys.version)
print("Version info:", sys.version_info)
```

Assignment 7: Import Functions Directly from a Module

Task: Import only `sqrt` and `pi` from the `math` module and use them.

```
from math import sqrt, pi

print("Square root of 49:", sqrt(49))
print("Value of pi:", pi)
```

Assignment 8: Creating a Custom Module

Task:

1. Create a file named `mymath.py` with a function `square(x)` that returns `x * x`.
2. Then import it in another script and use the function.

mymath.py

```
def square(x):
    return x * x
```

main.py

```
import mymath

print("Square of 8 is:", mymath.square(8))
```

Assignment 9: Module with Multiple Functions

Task:

1. Create a module named `geometry.py` with functions `area_circle(r)` and `perimeter_circle(r)`
2. Import and test both functions

geometry.py

```
import math

def area_circle(r):
    return math.pi * r * r

def perimeter_circle(r):
    return 2 * math.pi * r
```

main.py

```
import geometry

radius = 5
print("Area:", geometry.area_circle(radius))
print("Perimeter:", geometry.perimeter_circle(radius))
```

Assignment 10: `if __name__ == "__main__"` in a Module

Task: Create a module with some test code that only runs when the module is executed directly.

testmodule.py

```
def greet(name):
    return f"Hello, {name}!"

if __name__ == "__main__":
    print(greet("Tester"))
```

main.py

```
import testmodule

print(testmodule.greet("Alice"))
```

Lesson 22

I. What Is Oop

I. What is OOP in Python

1. Object-Oriented Programming (OOP)

OOP (**Object-Oriented Programming**) is a **way of structuring and organizing code** by grouping **data** and the **functions that work on that data** into single units called **objects**.

Think of **objects** as **real-world things**:

- A **Car** has:
  - **Attributes** → brand, color, speed

  - **Methods** → start(), stop(), accelerate()

- A **Bank Account** has:
  - **Attributes** → account number, balance
  - **Methods** → deposit(), withdraw(), check_balance()

Key Definitions

Term	Definition	Example
Class	A **blueprint** for creating objects.	`class Car:`
Object	An **instance** of a class.	`my_car = Car()`
Attribute	A variable that belongs to an object.	`self.color = "red"`
Method	A function inside a class.	`def start(self):`

Why use OOP?

1. **Organized Code** → Related data and functions are grouped together.
2. **Easier Maintenance** → If something changes, you only update it in one place.
3. **Reusability** → You can reuse classes across projects.
4. **Natural Modeling** → You can represent real-world entities as code objects.
5. **Scalable** → Works well for large projects with many interacting components.

Difference from Procedural Programming

So far, you've used **procedural programming**:

- Data (**variables**) and behavior (**functions**) are **separate**.
- This works fine for small scripts, but as your program grows, it becomes hard to manage.

Example: Procedural

```
# Procedural style
name = "John"
age = 30

def greet():
    print("Hello", name)

greet()
```

Problems:

- If we had multiple people, we'd need multiple separate variables.
- It's hard to keep data and behavior tied together.

Example: OOP

```
# OOP style
class Person:
    def __init__(self, name, age):
        self.name = name  # attribute
        self.age = age    # attribute

    def greet(self):      # method
        print(f"Hello, my name is {self.name}")

person1 = Person("John", 30)
person1.greet()
```

Advantages:

- The **data** (name, age) and **behavior** (greet) are **inside the same object**.
- You can create as many **Person** objects as you want without repeating code.

---

2. Names and Objects in Python

In Python, **everything is an object**.

When you create a variable:

`x = 10`

- **x** is a **name** that points to the object `10` (which is of type `int`).
- The **name** and the **object** are **different things**.

Example

```
a = [1, 2, 3]   # name a points to a list object
b = a           # b points to the same list object
b.append(4)

print(a)  # [1, 2, 3, 4] → both a and b point to the same object
```

- `a` and `b` are **names**.
- The list `[1, 2, 3, 4]` is the **object**.
- Changing the object affects all names pointing to it.

---

3. Scopes and Namespaces

A **namespace** is a **mapping between names and objects**. A **scope** is the **region of the code** where a name is visible.

Types of Scopes (LEGB Rule):

1. **Local (L)** → Inside the current function
2. **Enclosing (E)** → In enclosing functions (nested functions)
3. **Global (G)** → At the top level of the script
4. **Built-in (B)** → Predefined in Python

Example

```
x = "global"  # Global scope

def outer():
    x = "enclosing"

    def inner():
        x = "local"
        print(x)  # Local variable

    inner()

outer()  # Output: local
```

Namespace Example

```
# Global namespace
a = 5
print(globals())  # Shows all global variables

def func():
    # Local namespace
    b = 10
    print(locals())  # Shows all local variables

func()
```

---

4. Three New Object Types Introduced with Classes

When you define a class in Python, three new object types appear:

---

1. Class Object

- Created when you define a class.
- Used to create **instance objects**.

```
class MyClass:
    pass

print(type(MyClass))  # <class 'type'>
```

---

2. Instance Object

- Created when you call a class.

```
class Dog:
    pass

dog1 = Dog()  # Instance object
print(type(dog1))  # <class '__main__.Dog'>
```

---

3. Method Object

- A function inside a class that **belongs to an object**.
- When you call it, Python automatically passes the object as the first argument (`self`).

```
class Dog:
    def bark(self):
        print("Woof!")

dog1 = Dog()
dog1.bark()  # "Woof!"
print(dog1.bark)  # <bound method Dog.bark of <__main__.Dog object>>
```

---

5. Putting It All Together

Example showing:

- **Names and objects**
- **Scopes and namespaces**
- **Three new object types**

```
class Car:
    # Class attribute (belongs to all instances)
    wheels = 4

    def __init__(self, brand):
        # Instance attribute (unique to each object)
```

```
        self.brand = brand

    def drive(self):
        print(f"{self.brand} is driving.")

# Class object
print(Car)  # <class '__main__.Car'>

# Instance object
my_car = Car("Toyota")
print(my_car)  # <__main__.Car object>

# Method object
print(my_car.drive)  # Bound method
my_car.drive()

# Namespaces
print(my_car.__dict__)  # Instance namespace
print(Car.__dict__)     # Class namespace
```

Ii. Key Oop Concepts In Python

II. Key OOP Concepts in Python

A. Class

A **class** is a **blueprint** or **template** for creating objects. It defines:

- **What data** the objects will have (attributes)
- **What actions** the objects can perform (methods)

Think of a **class** like an **architect's blueprint** for a house:

- It describes the structure (rooms, walls, doors).
- But the blueprint itself is **not** a house — you must **create instances** (real houses) from it.

Example:

```
class Dog:
    pass  # Empty class for now

print(type(Dog))  # <class 'type'>
```

Here:

- `Dog` is a **class**.
- No attributes or methods are defined yet.

B. Object

An **object** is an **instance** of a class. It is **created** from the class blueprint and has **its own copy of the attributes** defined by the class.

Example:

```
class Dog:
    pass

dog1 = Dog()  # Create first object
dog2 = Dog()  # Create second object

print(dog1)  # <__main__.Dog object>
print(dog2)  # <__main__.Dog object>
```

- `dog1` and `dog2` are **different objects** created from the same `Dog` class.
- They are independent — changing one doesn't affect the other.

C. Attributes

Attributes are **variables that belong to an object**. They store **information** about that specific object.

We define attributes inside the **constructor** (`__init__` method).

Example:

```
class Dog:
    def __init__(self, name, age):
        self.name = name  # Instance attribute
        self.age = age

dog1 = Dog("Buddy", 3)
dog2 = Dog("Max", 5)

print(dog1.name)  # Buddy
print(dog2.age)   # 5
```

Here:

- `self.name` and `self.age` are **attributes**.
- `dog1` and `dog2` have **their own values** for these attributes.

D. Methods

Methods are **functions inside a class** that describe **what the object can do**. They always take **`self`** as the first parameter (which refers to the **current object**).

Example:

```python
class Dog:
    def __init__(self, name):
        self.name = name

    def bark(self):  # Method
        print(f"{self.name} says Woof!")

dog1 = Dog("Buddy")
dog2 = Dog("Max")

dog1.bark()  # Buddy says Woof!
dog2.bark()  # Max says Woof!
```

- `bark()` is a **method**.
- It works differently for each dog because it uses that object's `self.name`.

E. Constructor (`__init__` Method)

The **constructor** is a **special method** named `__init__` that runs **automatically** when you create an object.

It's used to:

- Initialize the object's attributes
- Set up default values if none are provided

Example:

```python
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

car1 = Car("Toyota", "Corolla")
car2 = Car("Honda", "Civic")

print(car1.brand)  # Toyota
print(car2.model)  # Civic
```

- Every time you create a `Car` object, Python automatically calls `__init__`.

F. Using `*args` and `**kwargs`

Sometimes, you don't know how many arguments will be passed when creating an object. Python lets you handle this with:

- `*args` → captures **multiple positional arguments**
- `**kwargs` → captures **multiple keyword arguments**

Example with `*args`

```
class NumberList:
    def __init__(self, *args):
        self.numbers = list(args)

num_list = NumberList(1, 2, 3, 4, 5)
print(num_list.numbers)  # [1, 2, 3, 4, 5]
```

Here:

- `*args` takes all numbers and stores them in `self.numbers`.

Example with `**kwargs`

```
class Person:
    def __init__(self, **kwargs):
        self.name = kwargs.get("name", "Unknown")
        self.age = kwargs.get("age", 0)
        self.city = kwargs.get("city", "Not specified")

p1 = Person(name="Alice", age=25, city="New York")
p2 = Person(name="Bob")

print(p1.name, p1.city)  # Alice New York
print(p2.name, p2.city)  # Bob Not specified
```

Here:

- `**kwargs` stores all keyword arguments in a dictionary.
- `.get()` is used to provide **default values** if the key is missing.

Best Practices Recap

1. **Classes** → Blueprints for objects
2. **Objects** → Actual things created from classes
3. **Attributes** → Variables inside objects
4. **Methods** → Functions inside objects
5. **Constructor (`__init__`)** → Automatically runs when creating an object
6. **`*args` & `**kwargs`** → Handle variable arguments flexibly

Exercise

6. Practical Examples

Pass

Begin with pass to create an class that we can instantiate.

```
some_number = 2
some_text = "Anna"

type(some_number)
type(some_text)
```

We are assigning attributres to our instance of this class.

```
class Item:
    pass

item1 = Item()
item1.name = "Phone"
item1.price = 100
item1.quantity = 5


print(type(item1)) # item
print(type(item1.name)) # str
print(type(item1.price)) # int
print(type(item1.quantity)) #int
```

We can create methods (we used to call them functions, but in classes they are called methods) and run against instances.

```
class Item:
    def calculate_total_price_test(self):
        pass

    def calculate_total_price(self, x, y):
        return x * y

item2 = Item()
item2.name = "Laptop"
item2.price = 1000
item2.quantity = 3
print(item2.calculate_total_price(item2.price, item2.quantity))
```

self passes this argument on to method, therefore Python uses this word to push propagate this attribute. With every new method, we need to have (!) at least one parameter. “self” is like a common convention

This class needs attributes, not that we need to instance these attributes every single time. There-

fore we create _____init_____ constructor (also known as magic method)

"init" is automatically created when object is instantiated.

```
class Item:
    def __init__(self):
        print("See, This is created")

item1 = Item()
item2 = Item()
```

So we want to get rid of "hard-coded" attributes for each instance. We will use init constructor

```
class Item:
    def __init__(self, name):
        print(f"An, Instance is created {name}")

item1 = Item("Phone")
#item2.name = "Prhone"
item2 = Item("LAptop")
#item2.name = "Laptop"
```

With dynamic attribute assignement using magic method _____init_____ we use self to create this attribute within the method, so that we do not need to instance it every time.

```
class Item:
    def __init__(self, name):
        self.name = name

item1 = Item("Phone")
item2 = Item("Laptop")
print(item1.name)
print(item2.name)
```

And let's do it for all the attributes. We call also assign a default value to the parameters in _____init_____ (e.g.: quantity=0)

```
class Item:
    def __init__(self, name, price, quantity):
        self.name = name
        self.price = price
        self.quantity = quantity


    def calculate_total_price(self, x, y):
        return x * y

item1 = Item("Phone", 100, 5)
item2 = Item("Laptop", 1000, 3)
```

```
print(item1.name)
print(item2.name)
print(item1.price)
print(item2.price)
print(item1.quantity)
print(item2.quantity)
```

We can also change now the “calculate_total_price” method.

```
class Item:
    def __init__(self, name, price, quantity):
        self.name = name
        self.price = price
        self.quantity = quantity


    def calculate_total_price(self):
        return self.price * self.quantity



item3 = Item("lala",10,23)
print(item3.name)
print(item3.calculate_total_price())
```

We need to validate the data types we type in!

```
class Item:
    def __init__(self, name, price, quantity):
        self.name = name
        self.price = price
        self.quantity = quantity


    def calculate_total_price(self):
        return self.price * self.quantity



item1 = Item("Phone","100",1) #making strings instead of integers
item2 = Item("Keyboard","10",5)

print(item1.calculate_total_price())
print(item2.calculate_total_price())
```

So we change it to and enforce also the validations using “assert”

```
class Item:
    def __init__(self, name: str, price: float, quantity: int):
        assert price >= 0, f"Price {price} is not negative"
```

```
        assert quantity >=0, f"Quantity must be positive or 0"
        self.name = name
        self.price = price
        self.quantity = quantity


    def calculate_total_price(self):
        return self.price * self.quantity



item1 = Item("Phone",100,1)
item2 = Item("Keyboard",10,5)

print(item1.calculate_total_price())
print(item2.calculate_total_price())
```

Accessing information from instance level and then on class level. Item1 and Item2 are instances of the class. and pay_rate is on class level!

```
class Item:
    pay_rate = 0.8
    def __init__(self, name: str, price: float, quantity: int):
        assert price >= 0, f"Price {price} is not negative"
        assert quantity >=0, f"Quantity must be positive or 0"
        self.name = name
        self.price = price
        self.quantity = quantity


    def calculate_total_price(self):
        return self.price * self.quantity



item1 = Item("Phone",100,1)
item2 = Item("Keyboard",10,5)

print(Item.pay_rate)
print(item1.pay_rate)
```

Find all the attributes that are available on instance and class level

```
class Item:
    pay_rate = 0.8
    def __init__(self, name: str, price: float, quantity: int):
        assert price >= 0, f"Price {price} is not negative"
        assert quantity >=0, f"Quantity must be positive or 0"
        self.name = name
```

```
        self.price = price
        self.quantity = quantity


    def calculate_total_price(self):
        return self.price * self.quantity

item1 = Item("Phone",100,1)

print(Item.__dict__)
print(item1.__dict__)
```

And finally, applying the discount (20%) using the pay_rate.

```
class Item:
    pay_rate = 0.8
    def __init__(self, name: str, price: float, quantity: int):
        assert price >= 0, f"Price {price} is not negative"
        assert quantity >=0, f"Quantity must be positive or 0"
        self.name = name
        self.price = price
        self.quantity = quantity


    def calculate_total_price(self):
        return self.price * self.quantity

    def apply_discount(self):
        self.price = self.price * self.pay_rate #Item.pay_rate or self.pay_rate

item1 = Item("Phone",100,1)
item1.apply_discount()
print(item1.price)


item2 = Item("Laptop", 1000, 3)
item2.pay_rate = 0.7
item2.apply_discount() #change self.pay_rate to Item.pay_rate to change the behaviour
print(item2.price)
```

Store all instances in object:

```
class Item:
    all = []
    pay_rate = 0.8
    def __init__(self, name: str, price: float, quantity: int):
        assert price >= 0, f"Price {price} is not negative"
        assert quantity >=0, f"Quantity must be positive or 0"
```

```
        self.name = name
        self.price = price
        self.quantity = quantity

        Item.all.append(self)


item1 = Item("apple",10,2)
item2 = Item("pear",70,4)
item3 = Item("egg",60,1)
item4 = Item("milk",40,4)

print(Item.all)

for inst in Item.all:
    print(inst.name)
```

we can use another magic function _____repr_____.

```
class Item:
    all = []
    pay_rate = 0.8
    def __init__(self, name: str, price: float, quantity: int):
        assert price >= 0, f"Price {price} is not negative"
        assert quantity >=0, f"Quantity must be positive or 0"
        self.name = name
        self.price = price
        self.quantity = quantity

        Item.all.append(self)

    def __repr__(self):
        return f"Item('{self.anem}, {self.price}')"


item1 = Item("apple",10,2)
item2 = Item("pear",70,4)
item3 = Item("egg",60,1)
item4 = Item("milk",40,4)

print(Item.all)
```

Example of classmethod

```
@classmethod
def instantiate_from_csv(cls):
    with open ('items.csv', 'r') as f:
        reader = csv. DictReader (f)
        items = list(reader)
```

```
    for item in items:
        Item(
            name=item get ( 'name'),
            price=float(item get('price')),
            quantity=int(item.get( 'quantity')),
        )


Item. instantiate_from_csv)
print(Item.all)
```

example of static method

```
@classmethod
def instantiate_from_csv(cls):
    with open ('items.csv', 'r') as f:
        reader = csv. DictReader (f)
        items = list(reader)

    for item in items:
        Item(
            name=item get ( 'name'),
            price=float(item get('price')),
            quantity=int(item.get( 'quantity')),
        )

@statiscmethod
def is_integer(num):
    if isinstance(num, float):
        return num.is_integer()
    elif isinstance(num, int):
        return True
    else:
        return False

print(Item.is_integer(7))   #7.0 = True; 7.5 = False; 7 = True
```

Difference between static and class method:

- use static - this should do something that has a relationship with the class but nothing to do that must be unique to the instance
- use class - this should also do something that has a relationshop with the class, but usually used to manipulate different structures of data to instantiate objects, like reading CSV file

Iii. Hands-On Example- Modeling Real-World Objects (Atm)

III. Hands-on Example: My First ATM Script :)

Example 1: ATM Machine

We start with a **simple ATM class** that can turn on and off. Later, we'll connect it to a **BankAccount**.

```
class ATM:
    def __init__(self, location, bank_name):
        self.location = location
        self.bank_name = bank_name

    def power_on(self):
        print(f"ATM at {self.location} ({self.bank_name}) is now ON.")

    def power_off(self):
        print(f"ATM at {self.location} ({self.bank_name}) is now OFF.")

atm1 = ATM("Copova ulica, Ljubljana", "NLB")
atm1.power_on()
atm1.power_off()
```

Output:

```
ATM at Copova ulica, Ljubljana (NLB) is now ON.
ATM at Copova ulica, Ljubljana (NLB) is now OFF.
```

Example 2: Bank Account

A **BankAccount** class stores account details and allows deposits/withdrawals.

```
class BankAccount:
    def __init__(self, account_holder, balance=0):
        self.account_holder = account_holder
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        print(f"Deposited ${amount}. New balance: ${self.balance}")

    def withdraw(self, amount):
        if amount > self.balance:
            print("Insufficient funds!")
        else:
            self.balance -= amount
            print(f"Withdrew ${amount}. New balance: ${self.balance}")


account1 = BankAccount("Tomaz K.", 500)
account1.deposit(200)
```

```
account1.withdraw(100)
account1.withdraw(700)
```

Output:

```
Deposited $200. New balance: $700
Withdrew $100. New balance: $600
Insufficient funds!
```

Example 3: Bank Client

A **Client** has a **BankAccount** and can interact with an ATM.

```
cclass Client:
    def __init__(self, name, account):
        self.name = name
        self.account = account

    def check_balance(self):
        print(f"{self.name}'s current balance: ${self.account.balance}")

    def use_atm(self, atm, action, amount=0):
        print(f"{self.name} is using the ATM at {atm.location}...")

        if action == "withdraw":
            self.account.withdraw(amount)
        elif action == "deposit":
            self.account.deposit(amount)
        elif action == "balance":
            self.check_balance()
        else:
            print("Invalid ATM action.")


TK_account = BankAccount("SI56 0201 00000 00001230", 1000)

client1 = Client("Tomaz Kastrun", TK_account)


atm1 = ATM("NLB Copova, Ljubljana", "NLB Banka")
atm1.power_on()


client1.use_atm(atm1, "withdraw", 200)
client1.use_atm(atm1, "deposit", 500)
client1.use_atm(atm1, "balance")

atm1.power_off()
```

Output:

```
ATM at NLB Copova, Ljubljana (NLB Banka) is now ON.
Tomaz Kastrun is using the ATM at NLB Copova, Ljubljana...
Withdrew $200. New balance: $800
Tomaz Kastrun is using the ATM at NLB Copova, Ljubljana...
Deposited $500. New balance: $1300
Tomaz Kastrun is using the ATM at NLB Copova, Ljubljana...
Tomaz Kastrun's current balance: $1300
ATM at NLB Copova, Ljubljana (NLB Banka) is now OFF.
```

---

4. Summary Table (Aligned to Bank Story)

Concept	Meaning	Example from Bank Story
Class	Blueprint for objects	`class BankAccount:`
Object	Instance of a class	`atm1 = ATM("NLB Copova, LJubljana", "NLB")`
Attribute	Variable inside a class	`self.balance`
Method	Function inside a class	`def deposit(self, amount)`
Constructor	Initializes an object	`def __init__(self, account_holder, balance=0)`
`*args`	Many positional args	`def __init__(self, *transactions)`
`**kwargs`	Many keyword args	`def __init__(self, **account_info)`

Some Ideas **extend this** even more:

- Multiple clients
- Multiple accounts
- Multiple ATMs
- `*args` & `**kwargs` usage
- Error handling for withdrawals
- Logging transactions to a file

Lesson 23

Atm Project Overview

ATM Project Overview

We will build a **fully functional ATM simulation** in Python that interacts with:

- **SQLite database** (for storing accounts, transactions, user info)
- **CSV log file** (for storing user actions: PIN entry, withdrawals, deposits, payments, etc.)
- **TXT error log file** (for exceptions and warnings)
- **Matplotlib & Pandas** for **data analysis and visualization**

Project Features

1. Database Setup & Structure

We'll use **SQLite** (no external DB needed) with the following tables:

SQL Schema

```
CREATE TABLE Clients (
    client_id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    pin TEXT NOT NULL,
    email TEXT,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE Accounts (
    account_id INTEGER PRIMARY KEY AUTOINCREMENT,
    client_id INTEGER NOT NULL,
    balance REAL DEFAULT 0.0,
    card_type TEXT DEFAULT 'debit',
    FOREIGN KEY (client_id) REFERENCES Clients(client_id)
);

CREATE TABLE Transactions (
    transaction_id INTEGER PRIMARY KEY AUTOINCREMENT,
    account_id INTEGER NOT NULL,
    merchant TEXT,
    amount REAL,
    transaction_type TEXT,
    datetime DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (account_id) REFERENCES Accounts(account_id)
);
```

We'll also have:

- `db_manager.py` → Handles all SQL queries & connections
- All queries will be parameterized to avoid **SQL injection**

---

2. Logging User Actions (CSV)

- **Actions logged:**
  - Login attempts (success/failure)
  - PIN entry attempts
  - Deposits / Withdrawals
  - Balance checks
  - Payments (merchant, amount)
  - Changes in account settings

CSV Format Example:

```
timestamp,client_id,action,details
2025-07-29 10:12:45,1,login_success,Entered correct PIN
2025-07-29 10:15:02,1,withdraw,Amount: 200
```

---

3. Error Logging (TXT)

- All runtime errors, database errors, and invalid input attempts will be stored in `error_log.txt`
- Example:

```
[2025-07-29 10:15:02] ERROR: Invalid withdrawal amount: -50
[2025-07-29 10:16:10] ERROR: Database connection lost
```

---

4. User Interaction (Console UI)

Menu Options:

1. **Login** (PIN check)
2. **Check Balance**
3. **Deposit Money**
4. **Withdraw Money**
5. **Make Payment** (merchant, amount, card type)
6. **Settings**
   - Change PIN
   - Change personal info
7. **Transaction History**
8. **Data Insights & Visualizations**
   - Balance over time
   - Payments by merchant
   - Statistics: mean, min, max, median, std deviation
9. **Exit**

---

5. Data Wrangling & Visualization

We will:

- Load **Transactions** table into Pandas
- Calculate:

  - Mean, Min, Max, Median, Std deviation of **transaction amounts**
  - Total spending per merchant

- Plot with **Matplotlib**:

  1. Balance over time line graph
  2. Payments per merchant bar chart
  3. Monthly spending trend

---

6. Input Validation & Error Handling

- PIN: must be 4 digits
- Withdrawal: must be positive & current balance
- Deposit: must be positive
- Merchant name: must be non-empty string
- Amount: must be numeric and positive
- Card type: only `"debit"` or `"credit"`

---

7. Structure in Classes & Functions

We will use **OOP** for clarity:

Classes

1. `ATM`
   - Manages login, menu navigation
2. `Client`
   - Stores user details, account links
3. `Account`
   - Deposit, withdraw, check balance
4. `Transaction`
   - Represents payments/withdrawals
5. `DatabaseManager`
   - Handles all SQL queries
6. `Logger`
   - Writes action logs (CSV) & error logs (TXT)
7. `DataAnalyzer`
   - Loads transactions into Pandas & produces statistics & plots

---

8. Extra Features for Realism

- Multiple clients and accounts in the database
- Simulation mode:
  - Randomly generate transactions for testing
- Option to export transaction history to **CSV**
- Detect **suspicious transactions** (large withdrawals, multiple same-merchant payments in short time)
- Password/PIN hashing (using `hashlib` for security)

---

9. Example Workflow

```
Welcome to Python ATM
Enter your PIN: ****
Login successful!

1. Check Balance
2. Deposit Money
3. Withdraw Money
4. Make Payment
5. Settings
6. View Statistics
7. Exit

> 3
Enter withdrawal amount: 200
Withdrawal successful. New balance: $800
Transaction saved to database and log file.
```

Atm Project – Full Code

ATM Project – Full Code Skeleton

** `db_manager.py` – Database Manager**

```python
import sqlite3
import os

class DatabaseManager:
    def __init__(self, db_path="data/atm.db"):
        os.makedirs("data", exist_ok=True)
        self.db_path = db_path
        self.conn = sqlite3.connect(self.db_path)
        self.create_tables()

    def create_tables(self):
```

```
        with self.conn:
            self.conn.execute("""
            CREATE TABLE IF NOT EXISTS Clients (
                client_id INTEGER PRIMARY KEY AUTOINCREMENT,
                name TEXT NOT NULL,
                pin TEXT NOT NULL,
                email TEXT,
                created_at DATETIME DEFAULT CURRENT_TIMESTAMP
            )
            """)
            self.conn.execute("""
            CREATE TABLE IF NOT EXISTS Accounts (
                account_id INTEGER PRIMARY KEY AUTOINCREMENT,
                client_id INTEGER NOT NULL,
                balance REAL DEFAULT 0.0,
                card_type TEXT DEFAULT 'debit',
                FOREIGN KEY (client_id) REFERENCES Clients(client_id)
            )
            """)
            self.conn.execute("""
            CREATE TABLE IF NOT EXISTS Transactions (
                transaction_id INTEGER PRIMARY KEY AUTOINCREMENT,
                account_id INTEGER NOT NULL,
                merchant TEXT,
                amount REAL,
                transaction_type TEXT,
                datetime DATETIME DEFAULT CURRENT_TIMESTAMP,
                FOREIGN KEY (account_id) REFERENCES Accounts(account_id)
            )
            """)

    def add_client(self, name, pin, email):
        with self.conn:
            cur = self.conn.execute(
                "INSERT INTO Clients (name, pin, email) VALUES (?, ?, ?)",
                (name, pin, email)
            )
            return cur.lastrowid

    def add_account(self, client_id, balance=0.0, card_type="debit"):
        with self.conn:
            cur = self.conn.execute(
                "INSERT INTO Accounts (client_id, balance, card_type) VALUES (?, ?, ?)",
                (client_id, balance, card_type)
            )
            return cur.lastrowid

    def get_client_by_pin(self, pin):
```

```
        cur = self.conn.execute("SELECT * FROM Clients WHERE pin=?", (pin,))
        return cur.fetchone()

    def get_account_by_client(self, client_id):
        cur = self.conn.execute("SELECT * FROM Accounts WHERE client_id=?", (client_id,))
        return cur.fetchone()

    def update_balance(self, account_id, new_balance):
        with self.conn:
            self.conn.execute(
                "UPDATE Accounts SET balance=? WHERE account_id=?",
                (new_balance, account_id)
            )

    def add_transaction(self, account_id, merchant, amount, transaction_type):
        with self.conn:
            self.conn.execute(
                "INSERT INTO Transactions (account_id, merchant, amount, transaction_type) VALU
                (account_id, merchant, amount, transaction_type)
            )

    def get_transactions(self, account_id):
        cur = self.conn.execute("SELECT * FROM Transactions WHERE account_id=?", (account_id,))
        return cur.fetchall()
```

** logger.py – Action & Error Logger**

```
import csv
import os
from datetime import datetime

class Logger:
    def __init__(self):
        os.makedirs("logs", exist_ok=True)
        self.action_log = "logs/actions_log.csv"
        self.error_log = "logs/error_log.txt"

    def log_action(self, client_id, action, details=""):
        with open(self.action_log, "a", newline="") as f:
            writer = csv.writer(f)
            writer.writerow([datetime.now(), client_id, action, details])

    def log_error(self, error_message):
        with open(self.error_log, "a") as f:
            f.write(f"{datetime.now()} - ERROR: {error_message}\n")
```

** data_analyzer.py – Data Analysis & Plots**

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

class DataAnalyzer:
    def __init__(self, db_manager):
        self.db = db_manager
        os.makedirs("plots", exist_ok=True)

    def _load_transactions_df(self, account_id):
        rows = self.db.get_transactions(account_id)
        df = pd.DataFrame(rows, columns=["id", "account_id", "merchant", "amount", "type", "dat
        df["datetime"] = pd.to_datetime(df["datetime"])
        return df

    def plot_balance_over_time(self, account_id):
        df = self._load_transactions_df(account_id)
        df["balance"] = df["amount"].cumsum()
        plt.figure()
        plt.plot(df["datetime"], df["balance"], marker="o")
        plt.title("Balance Over Time")
        plt.savefig("plots/balance_over_time.png")

    def plot_payments_by_merchant(self, account_id):
        df = self._load_transactions_df(account_id)
        merchant_totals = df.groupby("merchant")["amount"].sum()
        plt.figure()
        merchant_totals.plot(kind="bar")
        plt.title("Payments by Merchant")
        plt.savefig("plots/payments_by_merchant.png")

    def print_statistics(self, account_id):
        df = self._load_transactions_df(account_id)
        print("Mean:", df["amount"].mean())
        print("Min:", df["amount"].min())
        print("Max:", df["amount"].max())
        print("Median:", df["amount"].median())
        print("Std Dev:", df["amount"].std())
```

** atm.py – ATM Logic**

```
class ATM:
    def __init__(self, db_manager, logger, data_analyzer):
        self.db = db_manager
        self.logger = logger
```

```
        self.analyzer = data_analyzer

    def login(self):
        for _ in range(3):
            pin = input("Enter your 4-digit PIN: ")
            if not (pin.isdigit() and len(pin) == 4):
                print("Invalid PIN format!")
                continue
            client = self.db.get_client_by_pin(pin)
            if client:
                print(f"Welcome {client[1]}!")
                account = self.db.get_account_by_client(client[0])
                return client, account
            else:
                print("PIN not found.")
        print("Too many failed attempts.")
        return None, None

    def main_menu(self, client, account):
        while True:
            print("\n--- ATM MENU ---")
            print("1. Check Balance")
            print("2. Deposit Money")
            print("3. Withdraw Money")
            print("4. Make Payment")
            print("5. View Statistics")
            print("6. Exit")

            choice = input("Select option: ")
            if choice == "1":
                self.check_balance(account)
            elif choice == "2":
                self.deposit_money(account)
            elif choice == "3":
                self.withdraw_money(account)
            elif choice == "4":
                self.make_payment(account)
            elif choice == "5":
                self.view_statistics(account)
            elif choice == "6":
                break
            else:
                print("Invalid choice!")

    def check_balance(self, account):
        print(f"Your balance is: {account[2]:.2f}")
        self.logger.log_action(account[1], "CHECK_BALANCE")
```

```
def deposit_money(self, account):
    try:
        amount = float(input("Enter amount to deposit: "))
        if amount <= 0:
            raise ValueError("Deposit must be positive.")
        new_balance = account[2] + amount
        self.db.update_balance(account[0], new_balance)
        self.db.add_transaction(account[0], "Deposit", amount, "deposit")
        self.logger.log_action(account[1], "DEPOSIT", amount)
        print("Deposit successful.")
        account = self.db.get_account_by_client(account[1])
    except Exception as e:
        self.logger.log_error(str(e))
        print("Error during deposit.")

def withdraw_money(self, account):
    try:
        amount = float(input("Enter amount to withdraw: "))
        if amount <= 0 or amount > account[2]:
            raise ValueError("Invalid withdrawal amount.")
        new_balance = account[2] - amount
        self.db.update_balance(account[0], new_balance)
        self.db.add_transaction(account[0], "ATM Withdrawal", -amount, "withdrawal")
        self.logger.log_action(account[1], "WITHDRAW", amount)
        print("Withdrawal successful.")
        account = self.db.get_account_by_client(account[1])
    except Exception as e:
        self.logger.log_error(str(e))
        print("Error during withdrawal.")

def make_payment(self, account):
    try:
        merchant = input("Enter merchant name: ")
        amount = float(input("Enter amount: "))
        if amount <= 0 or amount > account[2]:
            raise ValueError("Invalid payment amount.")
        new_balance = account[2] - amount
        self.db.update_balance(account[0], new_balance)
        self.db.add_transaction(account[0], merchant, -amount, "payment")
        self.logger.log_action(account[1], "PAYMENT", f"{merchant}:{amount}")
        print("Payment successful.")
        account = self.db.get_account_by_client(account[1])
    except Exception as e:
        self.logger.log_error(str(e))
        print("Error during payment.")

def view_statistics(self, account):
    self.analyzer.print_statistics(account[0])
```

```
        self.analyzer.plot_balance_over_time(account[0])
        self.analyzer.plot_payments_by_merchant(account[0])
        print("Statistics & plots generated.")
```

`main.py` – Entry Point

```
from db_manager import DatabaseManager
from logger import Logger
from data_analyzer import DataAnalyzer
from atm import ATM

if __name__ == "__main__":
    db = DatabaseManager()
    logger = Logger()
    analyzer = DataAnalyzer(db)
    atm = ATM(db, logger, analyzer)

    # Optional: Add a test client
    if not db.get_client_by_pin("1234"):
        client_id = db.add_client("Tomaz K", "1234", "Tomaz@Tomaz.com")
        db.add_account(client_id, 500.0, "debit")

    client, account = atm.login()
    if client:
        atm.main_menu(client, account)
```

I.Short Review

SECTION 1: General Python Understanding

Lesson 1 – Python Basics

- Set up Python (IDE, Jupyter)
- Syntax, variables, data types
- Input/output (`print`, `input`)
- Simple programs & PEP standards

Lesson 2 – Control Flow

- Operators & counting
- Conditionals: `if`, `elif`, `else`
- Loops: `for`, `while`, `break`, `continue`

Lesson 3 – Control Flow (Exercises)

- Practice with loops and conditionals

Lesson 4 – Data Structures (Lists & Tuples)

- Creating & modifying lists
- Indexing, slicing, iteration

Lesson 5 – Data Structures (Exercises)

- List & tuple practice

Lesson 6 – Data Structures (Dictionaries & Sets)

- Key-value pairs in dictionaries
- Unique collections in sets

Lesson 7 – Data Structures (Exercises)

- Dictionary & set practice

Lesson 8 – Functions

- Defining & calling functions
- Arguments, return values, scope
- Lambda functions, decorators
- Regular expressions intro

Lesson 9 – Functions (Exercises)

- Function implementation practice

SECTION 2: Data Wrangling

Lesson 10 – Pandas & NumPy Basics

- DataFrames: loading, filtering, sorting

Lesson 11 – Pandas (Exercises)

- Basic data manipulation practice

Lesson 12 – Data Cleaning & Aggregation

- Handling missing values, duplicates
- Grouping, merging, pivoting

Lesson 13 – Data Cleaning (Exercises)

- Cleaning & summarizing data

Lesson 14 – SQL for Python Developers

- Relational database basics
- CRUD: `SELECT`, `INSERT`, `UPDATE`, `DELETE`
- Joins & relationships

Lesson 15 – Advanced SQL

- Subqueries, advanced joins, optimization

Lesson 16 – Integrating SQL with Python

- SQLAlchemy or similar
- Embedding SQL in Python
- Using Pandas with SQL results

Lesson 17 – SQL Integration (Exercises)

- Database queries from Python

SECTION 3: Data Visualization

Lesson 18 – Matplotlib Basics

- Reading & preparing data
- Different plot types
- Adding labels, legends, grids

Lesson 19 – Visualization (Exercises)

- Plot creation practice

SECTION 4: Modules, Classes & OOP

Lesson 20 – Error Handling & Modules

- `try/except/finally`
- Importing modules (`math`, `random`)

Lesson 21 – Error Handling (Exercises)

- Handling exceptions in programs

Lesson 22 – Introduction to OOP

- Classes & objects
- Attributes, methods, constructors
- Real-world object modeling

Lesson 24

Mini Project Assignement

Car Rental System – Mini assignment for certificate

---

1. Goals of the Assignment

Create a realistic **Car Rental System** that:

- Manages **multiple clients**, **multiple cars**, **rental and return transactions**
- Stores data in an **SQLite database**
- Logs **user actions** to a **CSV file**
- Logs **errors** to a **TXT file**
- Implements **input validation** and **error handling**
- Performs **data wrangling** to generate **client-specific statistics** and **visualizations**
- Is structured using **classes**, **methods**, and **proper data types**

---

2. Core Functionalities

1. Client Login / Registration

- **Login** with email or client ID
- Validate inputs
- Limit failed login attempts
- Allow new client registration with:
  - Name
  - Email
  - Phone Number
  - Driver's license (optional)
- Store client info in the DB

---

2. View Available Cars

- List all cars not currently rented
- Filter by category, price, or model
- Fetch from DB with proper queries

---

3. Rent a Car

- Input: Car ID, rental days

- Validation: Car must be available, days > 0
- Update:
  - Car status → rented
  - Insert rental record in `rentals` table
- Log the rental action in CSV
- Calculate cost = `daily_rate * days`

---

4. Return a Car

- Input: Rental ID or Car ID
- Calculate:
  - Total days
  - Late fees (if any)
- Update:
  - Car status → available
  - Rental record → completed
- Log the return

---

5. View My Rentals

- Show current and past rentals
- Sort by date, car, status

---

6. Update Client Settings

- Change name, phone, email
- Update DB records
- Log each action

---

7. View Statistics (Per Client)

- Show:
  - Total cars rented
  - Average rental duration
  - Most rented car
  - Total amount spent
- Visuals using **Matplotlib/Seaborn**:

- Rental durations over time
- Spending trend by month
- Pie chart of rental categories

---

3. Database Schema (SQLite)

`clients`

id | name | email | phone | registration_date |

`cars`

id | model | brand | category | daily_rate | is_rented (bool) |

`rentals`

| id | client_id | car_id | start_date | end_date | total_cost | status |

---

4. File Logging

User Actions – `actions_log.csv`

```
timestamp,client_id,action,details
2025-08-06 12:45:00,CL001,"RENT","Car ID: C003 for 3 days"
```

Error Logs – `error_log.txt`

```
[2025-08-06 13:15:21] ERROR: Invalid input for rental days. Must be integer > 0.
```

---

5. Input Validation & Error Handling

- Email format validation
- Check car availability
- Rental period > 0
- Catch all DB errors, file errors, etc.
- Centralized error logging

---

6. OOP Structure

Classes

```
class Client:
    def login()
    def register()
    def update_profile()
```

```
class Car:
    def list_available()
    def update_status()

class Rental:
    def rent_car()
    def return_car()
    def calculate_cost()
    def get_statistics()

class Database:
    def connect()
    def execute_query()
    def fetch()

class Logger:
    def log_action()
    def log_error()
```

---

7. Data Wrangling & Visualization

Use **Pandas + Matplotlib/Seaborn**:

- Load rentals per client
- Group by:
  - Car
  - Month
  - Duration
- Visualize:
  - Total amount spent
  - Popular cars
  - Rentals per month

. Optional Enhancements

- Admin panel to add/remove cars
- Car availability calendar
- Export client invoices to PDF
- Tiered membership (Gold, Silver, etc.)

8. Submission Instructions

What to Submit

You must submit a **single `.zip` file** or a **folder** containing:

- All your **Python source files** (`.py`)
- The **SQLite database file**: `data.db`
- Log files:
  - `actions_log.csv`
  - `error_log.txt`
- Optional: statistics images in `/visuals` folder
- A `README.txt` or `README.md` file with:
  - Short usage instructions
  - Required libraries
  - Entry point (e.g., `main.py`)

File Naming Convention

- Your `.zip` file or folder must be named:

  `yourname_yoursurname.zip`

  Example: `alex_smith.zip`

File Organization

It includes:

- `main.py` – Entry point
- `modules/` – Python classes for `Client`, `Car`, `Rental`, `Database`, `Logger`
- `database/data.db` – Empty SQLite database
- `logs/` – Placeholder for action and error logs
- `visuals/` – Placeholder for charts
- `README.md` – Usage instructions

How to Submit

Upload the `.zip` file to the **LMS platform**

12.08.2025, 23:59 (EOD) Late submissions **will not be accepted** unless Aneta's / Bojan's approval is given.