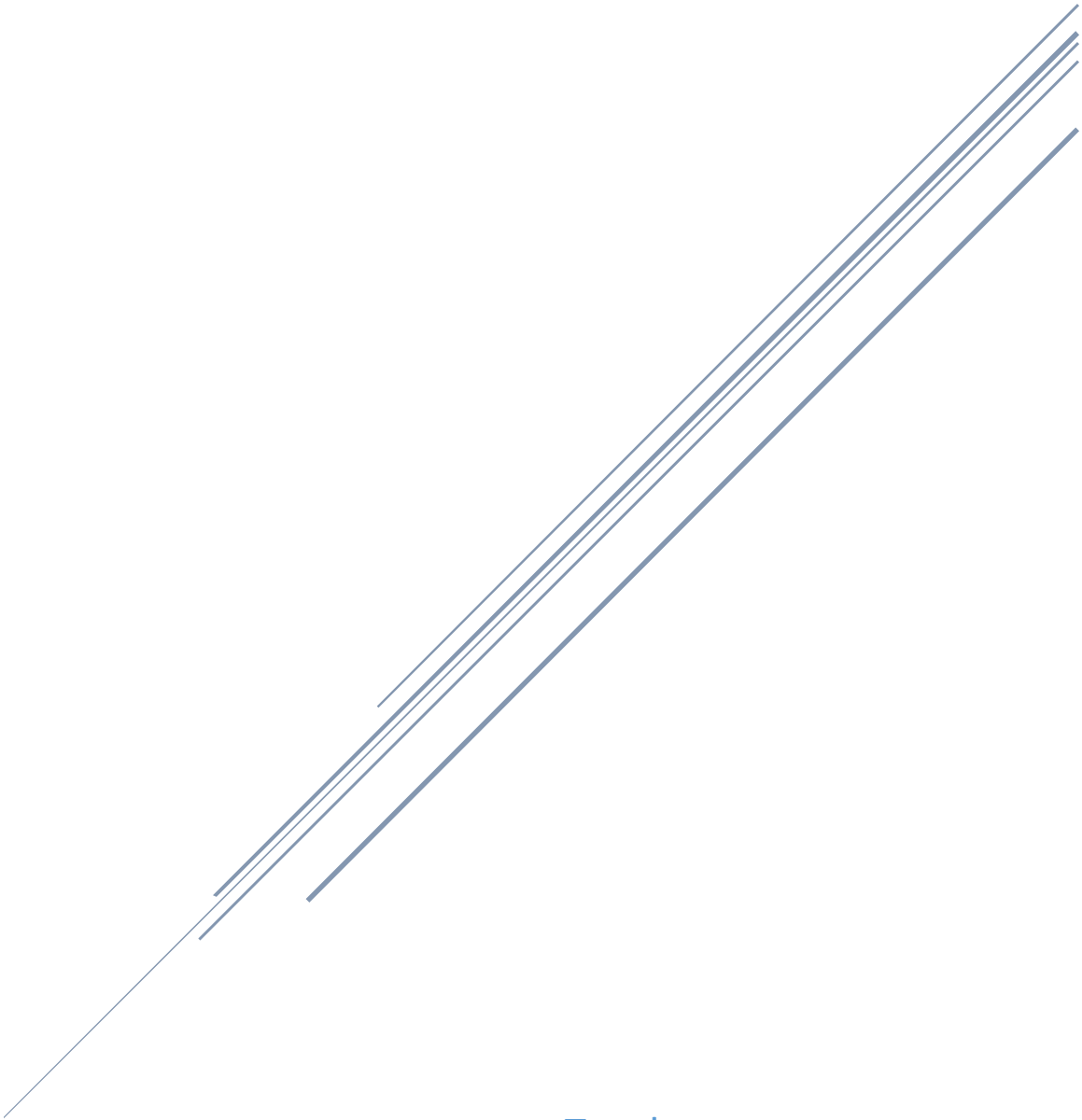
CODE AND SUMMARY OF CLASS 7

Sahir Ahmed Sheikh
Saturday (2 – 5)



Teachers:
Muhammad Bilal And Ali Aftab Sheikh

Code And Summary Of Class 7 – Saturday (2 – 5) | Quarter 3

Introduction

This document summarizes the topics covered in Class 7 of Quarter 3, held on Saturday from 2–5 PM. The session was conducted by Sir Ali Aftab Sheikh, Sir Muhammad Bilal, and Sir Aneeq Khatri, the Academic Excellence Coordinator (AEC) for the Governor House program. Sir Aneeq Khatri joined the class today with the purpose of observing the students' engagement and delivering essential Python concepts.

The class focused on:

- **Higher-Order Functions**: map() and filter().
- **Lambda Functions**: Anonymous, one-line functions in Python.
- **Object-Oriented Programming (OOP)**: Introduction to classes, objects, and the four pillars (Polymorphism, Inheritance, Abstraction, Encapsulation).

The session emphasized practical coding, conceptual understanding, and the importance of skills over degrees in the IT industry. Students were encouraged to engage actively, ask questions, and practice assignments to prepare for real-world challenges.

🏛 Topics Covered

1. Higher-Order Functions: map() and filter()

- **Concept**: Higher-order functions take other functions as arguments to process iterables efficiently, reducing manual looping.

- **Functions Covered**:
  - map(): Applies a function to each item in an iterable, returning a map object.
  - filter(): Filters items in an iterable based on a condition, returning a filter object.

- **Key Points**:
  - map() takes two arguments:
    1. A function (the work to be done).
    2. An iterable (the data to process).
  - filter() also takes a function and an iterable but returns only items where the function evaluates to True.
  - Both return **objects** (references), which must be converted to a list using list().
  - These functions simplify code, reducing the need for explicit loops.

Code Example: Using map()

```
# Define a function to square a number
def square(num):
    return num * num

# List of numbers
numbers = [1, 2, 3, 4, 5]

# Use map() to square each number
squared_numbers = list(map(square, numbers))

# Print the result
print(squared_numbers)  # Output: [1, 4, 9, 16, 25]
```

Explanation:

- square: A function that returns the square of a number.
- numbers: An iterable list [1, 2, 3, 4, 5].
- map(square, numbers): Applies square to each element, producing a map object.
- list(): Converts the map object to a list [1, 4, 9, 16, 25].

Code Example: Using filter()

```
# Define a function to square a number
def square(num):
    return num * num

# List of numbers
numbers = [1, 2, 3, 4, 5]

# Use map() to square each number
squared_numbers = list(map(square, numbers))

# Print the result
print(squared_numbers)  # Output: [1, 4, 9, 16, 25]
```

Explanation:

- greater_than_three: Returns True if a number is greater than 3, else False.
- filter(greater_than_three, numbers): Keeps only elements where the function returns True.
- list(): Converts the filter object to a list [4, 5].

Key Insight:

- Without map() or filter(), you'd need loops (e.g., 3–5 lines of code). These functions reduce code to **one line**, improving efficiency and readability.
- A third function, reduce(), was mentioned for self-exploration (used for aggregating data, e.g., summing a list).

2. Lambda Functions

- **Concept**: Lambda functions are anonymous (unnamed), one-line functions defined using the lambda keyword.
- **Syntax**: lambda arguments: expression
    - lambda: Keyword to define the function.
    - arguments: Parameters (e.g., x, y).
    - expression: A single expression (e.g., x + y).

- **Key Points**:
  - Used for short, simple tasks to avoid defining a full function with def.
  - Stored in variables to be called like regular functions.
  - Best for one-line logic; complex logic should use regular functions.

Code Example: Lambda Function for Addition

```
# Define a function to square a number
def square(num):
    return num * num

# List of numbers
numbers = [1, 2, 3, 4, 5]

# Use map() to square each number
squared_numbers = list(map(square, numbers))

# Print the result
print(squared_numbers)  # Output: [1, 4, 9, 16, 25]
```

Explanation:

- lambda x, y: x + y: Creates a function that adds two numbers.
- add: Stores the lambda function as a callable function.
- add(2, 5): Calls the function with arguments 2 and 5, returning 7.
- Compared to a regular function:

  ```
  def add(x, y):
      return x + y
  ```

  The lambda version is more concise (one line vs. two).

Code Example: Lambda with map()

```
# Use lambda with map to square numbers
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x * x, numbers))

# Print the result
print(squared_numbers)  # Output: [1, 4, 9, 16, 25]
```

Explanation:

- lambda x: x * x: Replaces the square function, squaring each number.
- map(): Applies the lambda to each element in numbers.
- Same output as the earlier map() example but without defining a separate function.

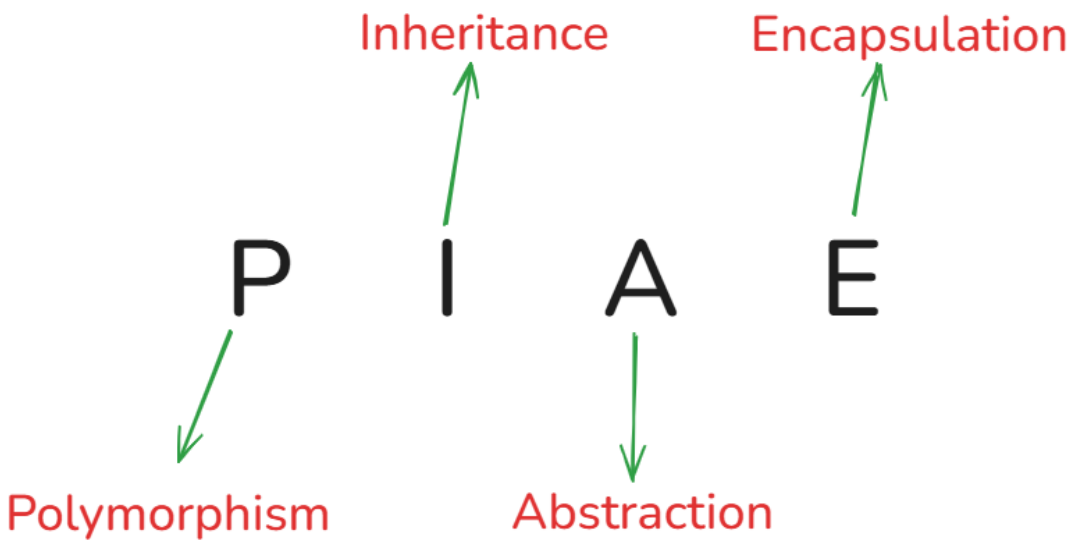
Key Insight:

- Use **lambda** for quick, one-line tasks (e.g., in map(), filter()).
- Use **regular functions** (def) for complex, multi-line logic.
- Lambda functions are similar to TypeScript's anonymous functions but use Python-specific syntax.

3. Introduction to Object-Oriented Programming (OOP)

- **Concept**: OOP is a programming paradigm that organizes code into **objects** created from **classes**, promoting modularity and reusability.
- **Key Terms**:
  - **Class**: A blueprint defining attributes (variables) and methods (functions).
  - **Object**: An instance of a class, created from the blueprint.
  - **Attributes**: Properties of a class (e.g., variables like user, password).
  - **Methods**: Functions defined in a class to perform actions.
- **Why OOP?**:
  - Breaks complex problems into smaller, reusable components.
  - Avoids **spaghetti code** (unstructured, tangled code).
  - Provides structure, making code easier to maintain and extend.
- **Four Pillars of OOP**:

  Mnemonic: PIAE (Polymorphism, Inheritance, Abstraction, Encapsulation), inspired by "Pakistan International Airlines" + E.



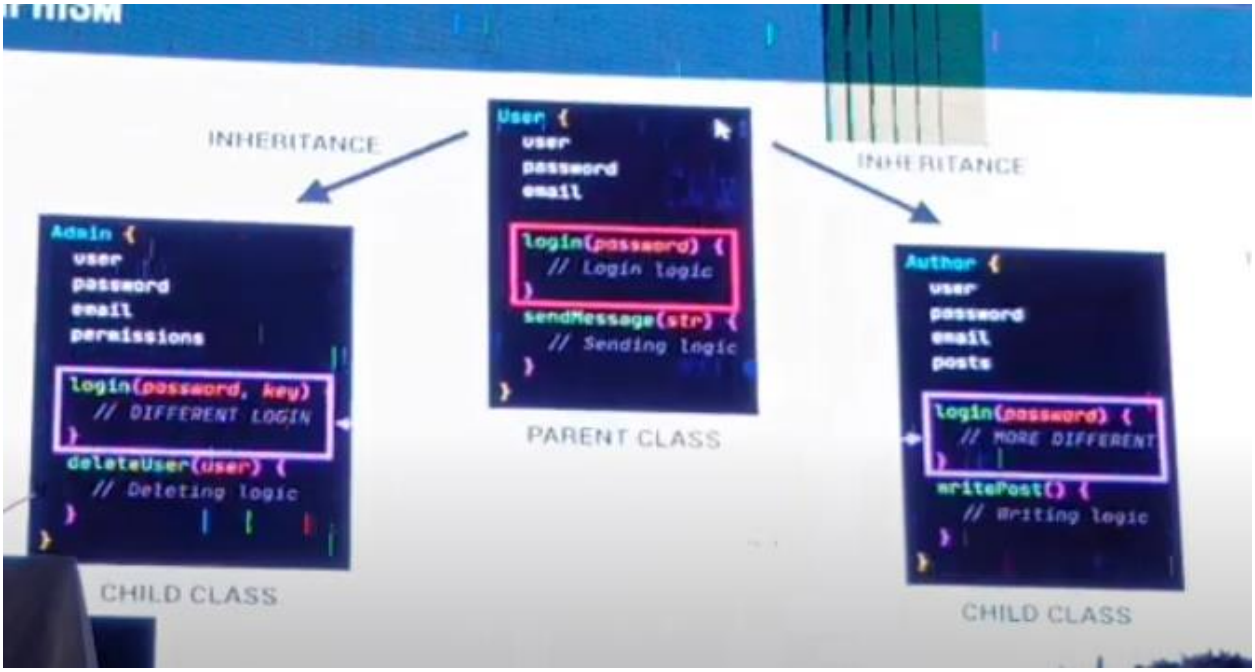
1. Polymorphism

Definition

Polymorphism means "many forms" (from Greek: *poly* = many, *morph* = form). It allows objects of different classes to be treated as objects of a common parent class, while each object can behave in its own unique way. In other words, the same method can perform different actions depending on the object calling it.

- **What It Does**: Polymorphism enables flexibility by letting you call a method on a parent class, but the actual behavior depends on the child class's implementation. This is achieved through **method overriding** (redefining a parent class method in a child class) or **method overloading** (less common in Python, but possible via default arguments).
- **Why It's Awesome**: It promotes code reusability and adaptability. You can write generic code that works with multiple object types without knowing their specific details.
- **Real-World Analogy**: Imagine a remote control with a "play" button. When you press it on a DVD player, it plays a movie. On a music player, it plays a song. The same button (method) behaves differently based on the device (object).

- **Types**:
  - **Compile-time (Static)**: Method overloading (not native in Python).
  - **Run-time (Dynamic)**: Method overriding (common in Python, via inheritance).
- **Benefits**:
  - Simplifies code by allowing a single interface for different behaviors.
  - Enhances maintainability and scalability.
- **When to Use**: When you want different classes to share a common interface but implement methods differently.



Coding Example: Polymorphism

Let's create a parent class Animal with a method speak(), and child classes Dog and Cat that override speak() to produce different sounds. We'll then treat all animals generically in a loop.

```python
# Parent class
class Animal:
    def speak(self):
        return "I make a sound"

# Child class: Dog
class Dog(Animal):
    def speak(self):  # Override parent's speak method
        return "Woof!"

# Child class: Cat
class Cat(Animal):
    def speak(self):  # Override parent's speak method
        return "Meow!"

# Function to demonstrate polymorphism
def make_animal_speak(animal):
    print(animal.speak())

# Create objects
dog = Dog()
cat = Cat()
generic_animal = Animal()

# Call speak() on different objects
make_animal_speak(dog)  # Output: Woof!
make_animal_speak(cat)  # Output: Meow!
make_animal_speak(generic_animal)  # Output: I make a sound

# Polymorphism in a loop
animals = [Dog(), Cat(), Animal()]
for animal in animals:
    print(animal.speak())  # Outputs: Woof!, Meow!, I make a sound
```

Explanation:

- **Classes**: Animal is the parent with a generic speak() method. Dog and Cat inherit from Animal and override speak() to return unique sounds.
- **Polymorphism**: The make_animal_speak function accepts any Animal object and calls speak(), but the output depends on the actual object (Dog, Cat, or Animal).
- **Loop**: The loop treats all objects as Animal but gets different behaviors, showcasing run-time polymorphism via method overriding.

- **Why It's Fun**: One function, multiple outcomes—like a DJ mixing different tracks with the same controls!

2. Inheritance

Definition

Inheritance allows a class (child or derived class) to inherit attributes and methods from another class (parent or base class). It promotes code reuse by letting child classes access and extend the functionality of parent classes without rewriting code.

- **What It Does**: A child class can use the parent's methods and attributes as-is, override them, or add new ones. It creates a hierarchical relationship (e.g., "is-a" relationship: a Dog is an Animal).
- **Why It's Awesome**: Reduces code duplication, makes maintenance easier, and models real-world hierarchies naturally.
- **Real-World Analogy**: Think of a family business. The child inherits the family's business practices (parent's methods) but can add their own innovations (new methods or overrides).
- **Types**:
  - **Single Inheritance**: One parent, one child.
  - **Multiple Inheritance**: One child, multiple parents (Python supports this).
  - **Multilevel Inheritance**: Grandparent → Parent → Child.
  - **Hierarchical Inheritance**: One parent, multiple children.
- **Benefits**:
  - Reuses existing code.
  - Supports extensibility (add new features in child classes).
  - Organizes code hierarchically.
- **When to Use**: When classes share common attributes/methods but need specialized behavior.

Coding Example: Inheritance

Let's create a parent class Vehicle with attributes like brand and a method drive(). A child class Car will inherit from Vehicle, add a num_doors attribute, and override drive().

```python
# Parent class
class Vehicle:
    def __init__(self, brand):
        self.brand = brand

    def drive(self):
        return f"{self.brand} vehicle is driving"

# Child class inheriting from Vehicle
class Car(Vehicle):
    def __init__(self, brand, num_doors):
        super().__init__(brand)  # Call parent's constructor
        self.num_doors = num_doors

    def drive(self):  # Override parent's drive method
        return f"{self.brand} car with {self.num_doors} doors is driving"

# Create objects
vehicle = Vehicle("Toyota")
car = Car("Honda", 4)

# Access attributes and methods
print(vehicle.brand)  # Output: Toyota
print(vehicle.drive())  # Output: Toyota vehicle is driving
print(car.brand)  # Output: Honda (inherited)
print(car.num_doors)  # Output: 4 (child-specific)
print(car.drive())  # Output: Honda car with 4 doors is driving
```

Explanation:

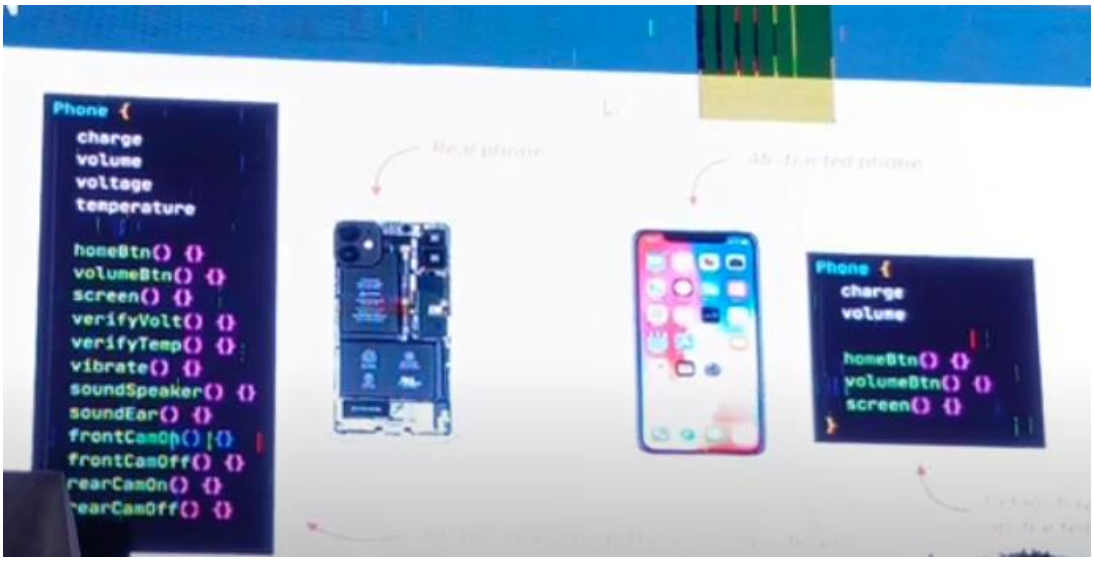
- **Classes**: Vehicle is the parent with a brand attribute and drive() method. Car inherits from Vehicle, adds num_doors, and overrides drive().
- **Inheritance**: Car uses super().__init__(brand) to inherit brand from Vehicle.
- **Access**: car can access brand (inherited) and num_doors (child-specific).
- **Why It's Fun**: It's like upgrading a basic vehicle into a sleek car with extra features, reusing the core design!

3. Abstraction

Definition

Abstraction hides complex implementation details and exposes only the necessary parts of an object to the user. It simplifies interaction by providing a clear interface while keeping the "how" under the hood.

- **What It Does**: Abstraction is achieved using **abstract classes** or **interfaces** (in Python, via the abc module). An abstract class defines methods that must be implemented by child classes, ensuring a consistent interface.
- **Why It's Awesome**: It reduces complexity, improves maintainability, and enforces a contract for subclasses to follow.
- **Real-World Analogy**: When you drive a car, you use the steering wheel and pedals (interface) without worrying about the engine's internals (hidden details).
- **How It's Done in Python**:
    - Use the abc module to create abstract base classes (ABCs).
    - Define **abstract methods** that child classes must implement.
- **Benefits**:
    - Simplifies user interaction with complex systems.
    - Enforces consistency across subclasses.
    - Hides unnecessary details, reducing errors.
- **When to Use**: When you want to define a template for classes but leave implementation details to subclasses.



Coding Example: Abstraction

Let’s create an abstract class Shape with an abstract method area(). Child classes Circle and Rectangle will implement area() differently.

```python
from abc import ABC, abstractmethod
import math

# Abstract base class
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass  # Abstract method, must be implemented by subclasses

# Child class: Circle
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):  # Implement abstract method
        return math.pi * self.radius ** 2

# Child class: Rectangle
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):  # Implement abstract method
        return self.width * self.height

# Create objects
circle = Circle(5)
rectangle = Rectangle(4, 6)

# Call area() on different shapes
print(f"Circle area: {circle.area():.2f}")  # Output: Circle area: 78.54
print(f"Rectangle area: {rectangle.area()}")  # Output: Rectangle area: 24
```

Explanation:

- **Abstract Class**: Shape inherits from ABC and defines area() as an abstract method using @abstractmethod. It cannot be instantiated directly.
- **Child Classes**: Circle and Rectangle implement area() with their specific formulas.
- **Usage**: You can call area() on any Shape object, but the implementation depends on the subclass.
- **Why It's Fun**: It's like giving artists a blank canvas (interface) and letting them create unique masterpieces (implementations)!

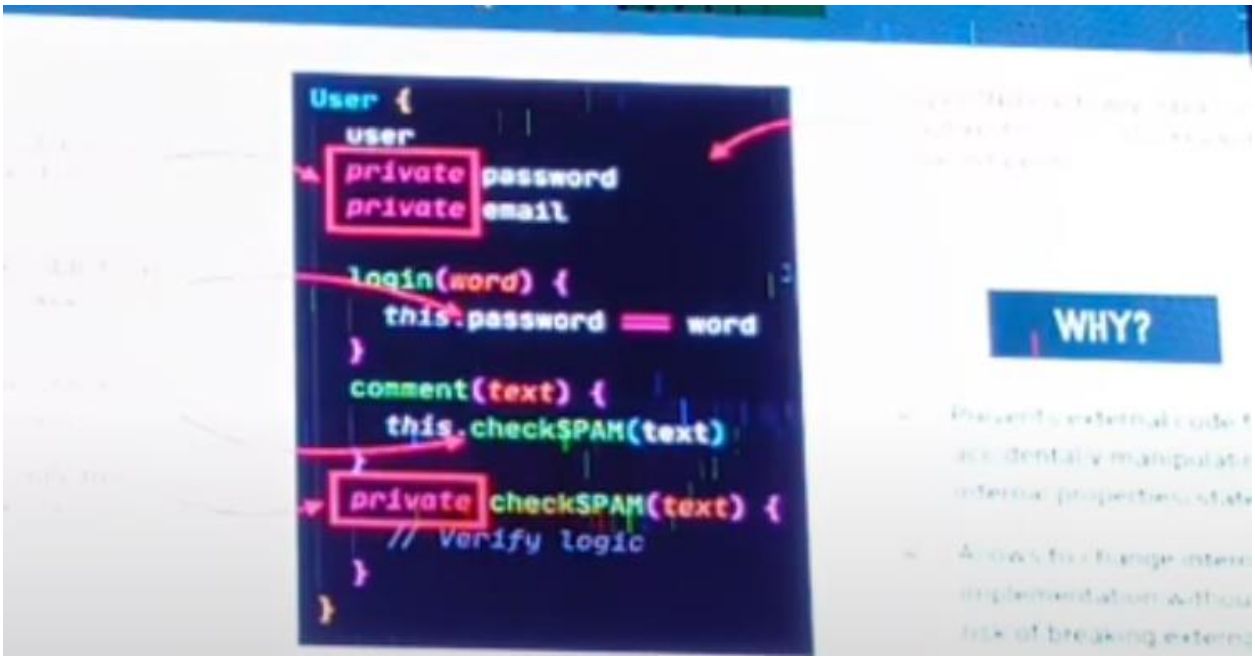
4. Encapsulation

Definition

Encapsulation bundles data (attributes) and methods that operate on that data into a single unit (class), while restricting direct access to some components. It protects an object's internal state and exposes only what's necessary through public methods.

- **What It Does**: Uses **access modifiers** (in Python, naming conventions like _ or __) to hide attributes and provide controlled access via **getters** and **setters**.
- **Why It's Awesome**: It ensures data integrity, prevents unauthorized changes, and makes code easier to maintain by controlling how data is accessed.
- **Real-World Analogy**: Think of a bank account. You can deposit or withdraw money through specific methods (teller or ATM), but you can't directly modify the balance (hidden data).
- **Python's Approach**:
  - **Public**: Attributes/methods accessible everywhere (e.g., name).
  - **Protected**: Indicated by _name (convention, not enforced).
  - **Private**: Indicated by __name (name mangling, restricts access).
- **Benefits**:
  - Protects data from accidental or malicious changes.
  - Allows controlled access and validation.
  - Enhances modularity by hiding implementation details.

- **When to Use**: When you want to safeguard an object's data and control how it's modified.



Coding Example: Encapsulation

Let's create a BankAccount class with a private balance attribute, accessible only through getter and setter methods that include validation.

```python
class BankAccount:
    def __init__(self, account_holder, initial_balance):
        self.account_holder = account_holder  # Public attribute
        self.__balance = initial_balance  # Private attribute

    # Getter for balance
    def get_balance(self):
        return self.__balance

    # Setter for balance (with validation)
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            return f"Deposited {amount}. New balance: {self.__balance}"
        else:
            return "Invalid deposit amount"

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            return f"Withdrew {amount}. New balance: {self.__balance}"
        else:
            return "Invalid or insufficient funds"

# Create an account
account = BankAccount("Ali", 1000)

# Access public attribute
print(account.account_holder)  # Output: Ali

# Access balance via getter
print(account.get_balance())  # Output: 1000

# Perform transactions
print(account.deposit(500))  # Output: Deposited 500. New balance: 1500
print(account.withdraw(200))  # Output: Withdrew 200. New balance: 1300
print(account.withdraw(2000))  # Output: Invalid or insufficient funds

# Try accessing private attribute directly (will fail)
# print(account.__balance)  # AttributeError
```

Explanation:

- **Class**: BankAccount has a public account_holder and a private __balance.

- **Encapsulation**: __balance is hidden and can only be accessed via get_balance(), deposit(), or withdraw().
- **Validation**: deposit() and withdraw() ensure only valid transactions modify __balance.
- **Access Attempt**: Trying to access __balance directly raises an error, showing data protection.
- **Why It's Fun**: It's like locking your treasure in a safe, giving out keys only to trusted methods!

Why This Is Exciting

The four pillars make OOP a powerful paradigm that's both structured and creative:

- **Polymorphism** lets you mix and match behaviors like a DJ spinning tracks.
- **Inheritance** builds on existing code like adding floors to a house.
- **Abstraction** simplifies complex systems like using a smartphone without knowing its circuits.
- **Encapsulation** protects your data like a vault, ensuring only authorized access.

Together, they enable modular, reusable, and maintainable code, turning programming into an art form where you craft elegant solutions to real-world problems.

Key Messages:

- **Skills Over Degrees**: The IT industry values problem-solving and practical skills.

- **Practice Assignments**: Engage with tasks, even if challenging, to build expertise.
- **Ask Questions**: Clarify doubts to master concepts like OOP, which are foundational for future topics (e.g., AI Agents).
- **Avoid Copy-Pasting**: Understand code to avoid issues in professional settings.

Additional Notes

- **Google Colab**: Used for coding exercises. Ensure familiarity with its interface.
- **OOP Importance**: Foundational for Quarter 4 (AI Agents) and industry projects.
- **Test Preparation**: Upcoming tests are critical; failing may result in course discontinuation.
- **Career Focus**: Stay focused on learning Python to seize opportunities (e.g., high-paying IT roles).

📚 Class Resources & Assignments

📌 Today's Class Code

The code for today's class can be accessed on Google Colab:
Class Code

🎯 Assignments

✅ *Assignment 1: Secure Data Encryption*

Work on the Secure Data Encryption project from the repository:
Secure Data Encryption – GitHub Repo

✔*Assignment 2: Complete Previous Assignments*

Make sure all previous assignments are completed and shared on LinkedIn, and also submitted via the class submission form.

⚠ Important Reminder

Please make sure to review and understand all 9 steps/topics before the exam. Due to limited class time, we couldn’t cover everything in class — so it's essential to go through the remaining topics on your own.

📅 Deadline: Before the next class

📌 Submission:

- Upload all assignments on LinkedIn
- Submit via the Assignment Submission Form

Stay consistent and keep learning!

🏁 Final Words:

The rest is up to **you** now. Learn actively. Engage consistently. Be confident in class participation—even wrong answers help you learn. Practice logic daily. Test will come soon.

Allah Hafiz — See you in the next class!

Thank You for Reading!

Hope you understood Class 7 well.

"Leadership is not about being in charge. It is about taking care of those in your charge" – *Simon Sinek*