coding temple

Python Basics

Module 4:

Lesson 3: OOP Principles

Intro

Hey Coding Templers! 🌟

Welcome to our Python OOP adventure! Ever wondered how sleek smartphones work? OOP in Python lets us create clean, efficient code just like that! Join us as we explore encapsulation, inheritance, and polymorphism with relatable analogies. Whether you're a beginner or a pro, our journey will boost your Python skills. Get ready to unlock the full potential of OOP and transform your code game! 🚀

Learning objectives

1. By the end of this lesson you should be able to:
2. Understand the concept of encapsulation and its significance in protecting and managing data within Python classes.
3. Apply the principles of inheritance in Python to create subclasses.
4. Illustrate the use of polymorphism in Python by writing code.
5. Distinguish between public, private, and protected attributes in Python classes to implement appropriate methods for accessing and modifying them.

coding temple

Understanding Encapsulation in OOP

It involves packaging both data and methods within our classes and managing their accessibility.

Example: Defining Classes in Python

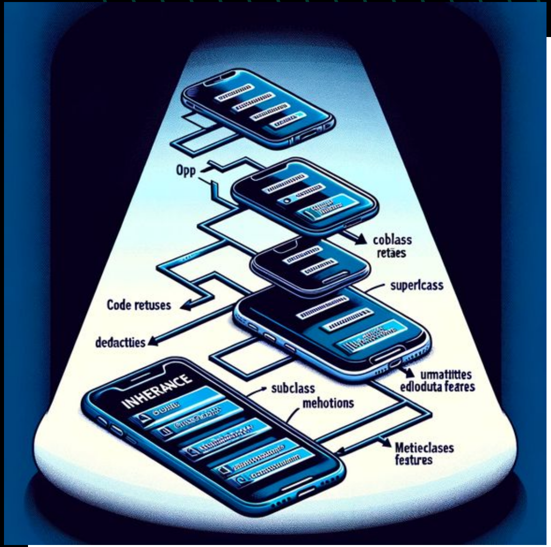
Let's create a `BankAccount` class to use getters and setters for both public and private attributes. This approach will showcase how we can manage and protect the state of our objects, ensuring a robust and secure implementation. We'll also add a few additional features to our class to make it more realistic and functional.

Objective: Implement a BankAccount class with a private balance and public account holder name. Use getters and setters to interact with these attributes and include methods for depositing and withdrawing funds.

The balance is a private attribute with a private setter, ensuring that the balance can only be modified through the deposit and withdrawal methods. The account holder's name is a public attribute, but we provide a getter and setter for it, following the encapsulation principle. This design maintains the integrity of the `BankAccount` class by controlling how its attributes are accessed and modified, akin to how a real bank account operates.

Understanding Inheritance

In programming, a class (known as a child or subclass) can inherit attributes and methods from another class (known as a parent or superclass), enabling code reuse and creating a hierarchical relationship.



Example: Creating Subclasses

How does it work?

Creating a subclass is like launching a new smartphone model. This new model (subclass) inherits the core functionalities of the older model (superclass) but can also have its unique features.

Why not check out a video to dive deeper into subclasses? It's a great way to get a better understanding!

In the following video, you will be able to see how `camera_phone`, an instance of `SmartCameraPhone`, demonstrates it can use both the inherited capabilities (like `make_call` from `Smartphone`) and its unique features (like `take_photo`).

Example: Overriding Parent Methods

How does it work?

Overriding parent methods is like customizing a feature in the new smartphone model. Although it inherits features from the older model, it can modify some features to suit its unique style.

Why not check out a video to dive deeper into Overriding Parent Methods? It's a great way to get a better understanding of how method overriding allows a subclass to provide a specific implementation for a method that is already defined in its superclass. It's a powerful feature of OOP that lets subclasses customize or extend the behavior of the superclass.

Exercise : Designing a BankAccount Inheritance Hierarchy

coding temple

01	Task	• **Base Class - BankAccount :** This is like the basic phone model, offering fundamental features like deposit and withdrawal. • **Subclasses - SavingsAccount and CheckingAccount**: These are specialized versions of BankAccount, each with additional features.
02	**Subclasses with Method Overriding**	We will now create two subclasses, each with its unique features. Additionally, we'll include method overriding to customize the behavior of these subclasses. • SavingsAccount Subclass : This subclass represents a savings account with an added interest rate feature. • CheckingAccount Subclass: This subclass represents a checking account with a transaction fee for each withdrawal.
03	Conclusion	In this exercise, the SavingsAccount and CheckingAccount subclasses add specific functionalities to the base BankAccount class and override the withdrawl method to suit their unique requirements. This shows how inheritance and method overriding work together to create flexible and customizable OOP structures.

Polymorphism in OOP

Polymorphism, refers to the ability of objects of different classes to respond to the same method call in their unique ways.

Method Overloading (Conceptual Understanding)

Method overloading is a form of polymorphism where multiple methods can have the same name but different parameters.
In Python, overloading allows a method to have default or variable parameters.

Now, let's see polymorphism in action with a simple Python exercise. We will create a base class and a couple of subclasses, each with the same method but behaving differently.

Best Practices in OOP

In this guide, we'll explore the best practices that help you write clean, efficient, and maintainable code using OOP principles:

1. Use **Encapsulation Wisely** to make your code more secure and maintainable.
2. Leverage **Inheritance Appropriately** to ensure a logical and intuitive class hierarchy.
3. Implement **Polymorphism for Flexibility** to allow objects of different classes to be treated uniformly.
4. Follow the **Single Responsibility Principle** to make your code more robust, easier to test, and simpler to maintain.
5. Prefer **Composition Over Inheritance** to provide greater flexibility and reduce tight coupling between classes.

Common Pitfalls in OOP

These are some "Common Pitfalls in OOP" that when addressed with awareness, can elevate your coding journey:

1. Overusing Inheritance
2. Ignoring Encapsulation
3. Misusing Polymorphism
4. Not Designing for Extensibility
5. Confusing Composition with Inheritance

In-Class Built-in functions Exercises: Hands-on Coding!

Hey there! Before we unveil our solutions, how about taking a shot at it yourself? Keep in mind that we've provided some broad strokes, so a detailed approach will help fine-tune the solution. But, this is all in good fun and a chance for you to flex those problem-solving muscles. Give it a go, and remember, it's all about learning and improving!

Exercise 1: Inventory Management System

Objective

Create an inventory management system for a retail store. This system will track products, manage stock levels, and handle sales transactions.

Problem Statement

A small retail store needs a simple inventory management system to keep track of its products, manage stock levels, and process sales. The system should allow adding new products, updating stock levels, and handling sales transactions, including generating receipts.

Hints

- Use a dictionary to map produc IDs to Product objects for easy retrieval.
- For sales transactions, loop through the list of Product objects to update stock levels and calculate total sales.
- Consider using a while loop for the main program to continuously offer options until the user decides to exit.
- Use try-except blocks to catch and handle exceptions like invalid inputs or operations on non-existent products.

Explanation

This program implements a basic inventory management system. The Product class encapsulates product-related data and operations. Users can add products, update stock levels, and process sales, with the system automatically adjusting inventory and providing transaction receipts. The program demonstrates the principles of encapsulation, along with the use of lists, dictionaries, and basic error handling to create a user-friendly, robust system.

Exercise 1: Inventory Management System - Instructions

Instructions

1. Create a Product class to represent individual products. Each product should have attributes like product ID, name, price, and quantity.
2. Implement encapsulation in the Product class, ensuring that direct access to attributes is restricted and managed through getters and setters.
3. Develop a main program that allows users to:
    - Add new products to the inventory.
    - Update stock levels for existing products.
    - Process sales transactions, updating stock levels, and printing a sales receipt.
4. Handle incorrect inputs or actions gracefully using try-except blocks.
5. Store the inventory as a list of Product objects.
6. Optionally, use a text file to load initial inventory data and save changes.

Exercise 2: Library Management System

Objective

Develop a library management system that manages book inventory and user checkouts. This system should handle adding new books, checking books in and out, and tracking user loans.

Problem Statement

A community library requires a system to keep track of its books and the loans made to library members. The system should allow librarians to add new books, check books in and out, and maintain a record of who has borrowed which book.

Hints

- Use the ISBN as a unique identifier for each book in the library's record-keeping.
- In the main program, use a loop to continually offer options until the librarian chooses to exit.
- For checking books in and out, update the availability status and keep a record of the user who has borrowed the book.

Explanation

This exercise implements a simple library management system. The Book class encapsulates details about each book, including its availability. The main program offers options to add new books, check them out to users, and check them in when returned. The system uses a dictionary to track all books and a separate dictionary to track current loans. This exercise demonstrates encapsulation, error handling, and the use of collections (lists/dictionaries) to manage complex relationships in a real-world application.

Exercise 2: Library Management System - Instructions

Instructions

1. Create a Book class with attributes like title, author, ISBN, and availability status.
2. Implement encapsulation in the Book class to ensure that direct access to attributes is controlled through getters and setters.
3. Develop a main program that allows librarians to:
   - Add new books to the library.
   - Check out books to users (updating availability status).
   - Check in books from users.
4. Maintain a record (dictionary or list) of all books in the library and their status.
5. Handle input errors or invalid actions using try-except blocks.
6. Utilize a list or set to track users who have currently borrowed books.
7. Optionally, implement text file handling to save and load library data.

Exercise 3: Vehicle Rental System

Objective

Build a vehicle rental system that can manage different types of vehicles, such as cars and bikes. The system should be able to handle vehicle rentals, returns, and display available vehicles.

Problem Statement

A rental service needs a system to manage its fleet of various types of vehicles. Each vehicle type has unique attributes and rental rates. The system should allow the staff to rent out vehicles, accept returns, and provide an overview of the available fleet.

Hints

- In the Vehicle class, use a boolean attribute to track whether a vehicle is available or not.
- Override methods in Car and Bike if their rental process differs from the base class.
- In the main program, use a loop with a menu to let the user choose different actions (e.g., rent, return, view available vehicles).

Explanation

This exercise demonstrates the use of inheritance in a vehicle rental system. The base class Vehicle contains attributes and methods common to all vehicles. The Car and Bike subclasses extend Vehicle by adding specific attributes like car_type and bike_type. The system can add new vehicles, rent them out, accept returns, and display available vehicles. The use of inheritance simplifies the management of different vehicle types, showcasing how to create a flexible and scalable system.

Exercise 3: Vehicle Rental System - Instructions

Instructions

1. Create a base class `Vehicle` with common attributes like make, model, and rental rate.
2. Develop subclasses `Car` and `Bike` that inherit from `Vehicle` and have additional specific attributes.
3. Implement methods in the classes for renting and returning vehicles, adjusting availability as needed.
4. Create a main program that:
   - Adds new vehicles to the fleet.
   - Handles renting and returning vehicles.
   - Displays available vehicles for rent.
5. Use a list to store the fleet of vehicles and track their availability status.
6. Apply error handling for situations like trying to rent out an already rented vehicle.

Exercise 4: Smart Home Automation System

Objective

Develop a smart home automation system that can control various smart devices. Each device type should have unique functionalities, but all share some common controls.

Problem Statement

In a modern smart home, various devices like lights, thermostats, and security cameras need to be managed through a centralized system. While each device has specific functions, they all share some common controls such as turning on/off and resetting.

Hints

- In the SmartDevice class, use print statements to simulate turning on/off and resetting the device.
- Override these methods in the subclasses to reflect the specific actions of each device.
- Use a list or dictionary to store and manage multiple devices in the main program.

Explanation

In this exercise, the SmartDevice class represents a generic smart device with basic functionalities. The subclasses SmartLight, SmartThermostat, and SmartCamera each override some of these methods to provide specific behaviors, demonstrating method overriding. For example, the turn_on method behaves differently for a light, a thermostat, and a camera, reflecting their unique functions in a smart home system. This exercise showcases how method overriding can be used to implement polymorphic behavior in an object-oriented system, allowing for more flexible and intuitive interactions with different types of objects.

Exercise 4: Smart Home Automation System - Instructions

Instructions

1. Create a base class SmartDevice with common methods like turn_on, turn_off, and reset.
2. Develop subclasses for specific devices: SmartLight, SmartThermostat, and SmartCamera. Each subclass should override some of the base class methods to implement device-specific behavior.
3. Implement a main program that allows users to interact with and control different smart devices.
4. Use method overriding to customize the behavior of common methods for each specific device.
5. Handle any user input errors or invalid operations using try-except blocks.

Exercise 5: Shape Area Calculator

Objective

Create a program to calculate the area of different geometric shapes, demonstrating polymorphism in Python.

Problem Statement

A graphic design software needs a feature to calculate the area of various shapes like circles, rectangles, and triangles. While each shape calculates its area differently, the software should handle them through a uniform interface.

Hints

- In subclasses, use appropriate formulas to calculate the area of each shape.
- Utilize the isinstance function to ensure that the object passed to the area calculation function is a subclass of Shape.
- Consider using Python's math module for mathematical operations, especially for the circle's area calculation.

Explanation

This exercise demonstrates polymorphism through a shape area calculator. The Shape class provides a common interface with the calculate_area method. Each subclass (Circle, Rectangle, Triangle) overrides this method to calculate its specific area. The get_area function, using polymorphism, can calculate the area of any shape object passed to it, regardless of the shape's specific type. This setup allows for easy extension to include more shapes in the future without modifying the existing code structure, showcasing the flexibility and scalability benefits of polymorphism in object-oriented design.

Exercise 5: Shape Area Calculator - Instructions

Instructions

1. Define a base class Shape with a method calculate_area (without implementation).
2. Create subclasses Circle, Rectangle, and Triangle, each overriding the calculate_area method.
3. Implement a function in the main program to accept a Shape object and call its calculate_area method.
4. Use polymorphism to allow the same function to calculate the area for any shape.
5. Handle invalid shape dimensions or operations using try-except blocks.

Wrapping It Up: Embracing the Power and Elegance of OOP in Python

On this journey, we've learned about:

- Encapsulation, likened to the user-friendly interface of a smartphone.
- Inheritance, likened to passing features from older to newer smartphone models.
- Polymorphism, visualized through the idea of a universal app working differently on various devices.

We trust you had a blast with this lesson! Catch you later, and happy coding!

Assignments

"Ready for the next adventure in your learning journey? Let's dive into the exciting world of assignments on the next slide! 🚀📚"