Chapter 1: Fundamentals of Object-Oriented Programming (OOP)

Before we start coding, let's build a solid conceptual foundation.

1.1 What is Object-Oriented Programming (OOP)?

Object-Oriented Programming (OOP) is a programming paradigm that organizes code around objects rather than functions. Objects represent real-world entities and have properties (data) and behaviors (methods).

Key Idea:

Instead of writing a series of instructions, we create objects that interact with each other.

Example (Real-World Analogy)

- A **car** is an object.
- It has **properties** (color, brand, model, speed).
- It has **behaviors** (start, stop, accelerate).

In Python, we define such objects using **classes**.

1.2 Why Use OOP?

OOP helps in:
✅ **Organizing Code** – Breaks down large programs into smaller, manageable objects.
✅ **Reusability** – You can reuse code through inheritance.
✅ **Scalability** – Easy to expand and maintain.
✅ **Encapsulation** – Protects data from unintended changes.
✅ **Abstraction** – Hides unnecessary details from the user.

Without OOP, large programs can become messy and hard to manage.

1.3 Four Pillars of OOP

1. Encapsulation 🛡

- **Definition:** Binding data (variables) and methods (functions) together in a single unit (class).
- **Example:** A car's internal engine mechanism is hidden from the user.

2. Abstraction 🎭

- **Definition:** Hiding unnecessary details and showing only essential features.
- **Example:** When driving a car, you don't need to know how the engine works; you just press the accelerator.

3. Inheritance 👨‍👩‍👧

- **Definition:** Allows one class (child) to inherit properties and methods from another class (parent).
- **Example:** A sports car inherits properties from a general car.

4. Polymorphism 🔀

- **Definition:** One function or method behaving differently based on input.
- **Example:** A person behaves differently as a student in school and as a gamer at home.

1.4 Understanding Objects and Classes

Objects:

- An **object** is an instance of a class.
- It has **attributes** (data) and **methods** (functions).

Classes:

- A **class** is a blueprint for creating objects.
- It defines what attributes and methods an object will have.

Analogy

Think of a **class** as a **blueprint** for a house, and the **objects** as different houses built using that blueprint.

1.5 Practical Example (First Code in OOP)

Now, let's implement these concepts in Python.

```
python
# Defining a Class
class Car:
    def __init__(self, brand, model, color):
        self.brand = brand  # Attribute
        self.model = model
        self.color = color
```

```
    def show_details(self):  # Method
        print(f"Car: {self.brand} {self.model}, Color: {self.color}")

# Creating Objects (Instances of the Class)
car1 = Car("Toyota", "Corolla", "Black")
car2 = Car("Honda", "Civic", "Red")

# Using Methods
car1.show_details()
car2.show_details()
```

Output:

```
Car: Toyota Corolla, Color: Black
Car: Honda Civic, Color: Red
```

Common Mistakes to Avoid

🚫 Forgetting to use `self` inside methods.
🚫 Not initializing object attributes in `__init__`.
🚫 Using class attributes when instance attributes are needed.

Chapter 2: Classes and Objects in Python

In the first chapter, we learned the fundamental concepts of OOP. Now, we will dive deep into **classes and objects**, understanding how to use them effectively.

2.1 What is a Class?

A **class** is a **blueprint or template** that allows us to create one or more **objects**.

Why Do We Need Classes?

- If you want to build a car, you don't design it from scratch every time. Instead, you create **a blueprint (class)**.
- Then, using that blueprint, you can create multiple **cars (objects)**.

Basic Syntax

```
python
class ClassName:
    # Constructor Method (Called Automatically)
    def __init__(self, attribute1, attribute2):
        self.attribute1 = attribute1
        self.attribute2 = attribute2

    # Method (Function inside a class)
    def method_name(self):
        print("This is a method.")
```

2.2 What is an Object?

An **object is a real-world instance of a class**.

- If a **class** is the design of a mobile phone, then an **object** is an actual mobile phone built using that design.

How to Create an Object?

```
python
object_name = ClassName(value1, value2)
```

2.3 First Program: Creating a Class and an Object

```
python
# Defining a class
class Car:
    def __init__(self, brand, model, color):
        self.brand = brand  # Attribute
        self.model = model
        self.color = color

    def show_details(self):  # Method
        print(f"Car: {self.brand} {self.model}, Color: {self.color}")

# Creating objects
car1 = Car("Toyota", "Corolla", "Black")
car2 = Car("Honda", "Civic", "Red")

# Calling methods
car1.show_details()
car2.show_details()
```

Output:

```
Car: Toyota Corolla, Color: Black
Car: Honda Civic, Color: Red
```

2.4 `__init__()` Method: What is a Constructor?

`__init__()` is a **constructor method** that gets called **automatically** when a new object is created.

```
python
class Mobile:
    def __init__(self, brand, price):
        self.brand = brand
        self.price = price
        print(f"{brand} mobile has been created!")

# Creating objects will automatically call `__init__()`
m1 = Mobile("Samsung", 20000)
m2 = Mobile("iPhone", 100000)
```

Output:

```
Samsung mobile has been created!
iPhone mobile has been created!
```

2.5 What is a Method?

A **method is a function inside a class** that processes the data of the object.

```
python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):  # Method
        print(f"Hello, my name is {self.name} and I am {self.age} years 
old.")

# Creating objects
p1 = Person("Alice", 25)
p2 = Person("Bob", 30)

# Calling methods
p1.greet()
p2.greet()
```

Output:

```
Hello, my name is Alice and I am 25 years old.
Hello, my name is Bob and I am 30 years old.
```

2.6 What is `self`, and Why Do We Need It?

◈ `self` helps each **object maintain its own data**.
◈ It **connects object variables and methods inside a class**.

```
python
class Laptop:
    def __init__(self, brand, ram):
        self.brand = brand
        self.ram = ram

    def show(self):
        print(f"Laptop: {self.brand}, RAM: {self.ram}GB")

# Creating objects
laptop1 = Laptop("Dell", 8)
laptop2 = Laptop("HP", 16)

# Calling methods
laptop1.show()
laptop2.show()
```

Output:
```
Laptop: Dell, RAM: 8GB
Laptop: HP, RAM: 16GB
```

⚠ **Common Mistakes to Avoid:**
🚫 Forgetting to use `self` inside methods.
🚫 Using `brand = brand` instead of `self.brand = brand` inside `__init__()`.

Chapter 3: Encapsulation and Data Hiding in Python

In this chapter, we will explore **Encapsulation**, one of the core principles of OOP. We will learn how to protect data, control access, and implement **data hiding** using access modifiers.

3.1 What is Encapsulation?

Encapsulation is the process of **bundling data (variables) and methods (functions) inside a class** while **restricting direct access** to some details.

Why is Encapsulation Important?

✅ Protects **sensitive data** from accidental modification.
✅ **Restricts direct access** to important attributes.
✅ Ensures **better control** over how data is changed.
✅ Makes code **more maintainable and scalable**.

3.2 Access Modifiers: Public, Protected, and Private

Python has three types of **access modifiers** to control access to class attributes and methods.

Modifier	Syntax	Access Level
Public	`self.var`	Accessible from anywhere
Protected	`self._var`	Accessible within the class and subclasses
Private	`self.__var`	Accessible only inside the class

3.3 Public Members

◈ **Public members** can be accessed and modified from anywhere.
◈ By default, all class attributes and methods are **public** unless specified otherwise.

```
python
class Car:
    def __init__(self, brand, model):
        self.brand = brand  # Public attribute
        self.model = model  # Public attribute

    def show(self):  # Public method
        print(f"Car: {self.brand} {self.model}")

# Creating an object
car1 = Car("Toyota", "Corolla")

# Accessing public attributes
print(car1.brand)  # ✅ Allowed
print(car1.model)  # ✅ Allowed

# Modifying public attributes
car1.model = "Camry"  # ✅ Allowed
car1.show()
```

Output:

```
Toyota
Corolla
Car: Toyota Camry
```

⚠ **Problem:** Public attributes can be modified from outside, which may lead to unintended changes.

3.4 Protected Members

◈ **Protected members** are **indicated by a single underscore _** (e.g., `_attribute`).
◈ They **should not** be accessed directly, but they **can** be accessed if necessary.

```
python
class Laptop:
    def __init__(self, brand, price):
        self.brand = brand
        self._price = price  # Protected attribute

    def show(self):
        print(f"Laptop: {self.brand}, Price: {self._price}")

# Creating an object
laptop1 = Laptop("Dell", 800)
```

```
# Accessing protected attributes (not recommended)
print(laptop1._price)  # ⚠ Technically possible but not recommended

# Modifying protected attributes (not recommended)
laptop1._price = 900
laptop1.show()
```

Output:

```
800
Laptop: Dell, Price: 900
```

◈ **Protected members** can be accessed from **subclasses**, which makes them useful in inheritance.
◈ However, they should be treated as **"not to be accessed directly"** unless absolutely necessary.

3.5 Private Members

◈ **Private members** are **indicated by double underscores __** (e.g., `__attribute`).
◈ They **cannot** be accessed directly from outside the class.

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

    def show_balance(self):
        print(f"Balance: ${self.__balance}")

# Creating an object
account = BankAccount(1000)

# Trying to access a private attribute
# print(account.__balance)  # ✕ This will cause an error

# Using a public method to access private data
account.show_balance()
```

Output:

```
Balance: $1000
```

⃠ **Directly accessing `__balance` will cause an AttributeError.**
☑ The only way to access it is through a **method inside the class**.

3.6 Getters and Setters: Accessing Private Attributes

Since **private attributes** cannot be accessed directly, we use **getter and setter methods** to **read and modify private data safely**.

Getter Method (To Read Private Data)

```python
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance

    def get_balance(self):  # Getter method
        return self.__balance

# Creating an object
account = BankAccount(2000)

# Accessing private data using the getter method
print(account.get_balance())  # ✅ Allowed
```

Setter Method (To Modify Private Data)

```python
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance

    def get_balance(self):
        return self.__balance

    def set_balance(self, amount):  # Setter method
        if amount >= 0:
            self.__balance = amount
        else:
            print("Invalid balance amount!")

# Creating an object
account = BankAccount(3000)

# Modifying private data using the setter method
account.set_balance(3500)  # ✅ Allowed
print(account.get_balance())

account.set_balance(-500)  # ❌ Invalid modification
```

Output:

```
3500
Invalid balance amount!
```

3.7 Name Mangling: Accessing Private Members Indirectly

◈ Python uses **name mangling** to modify private attributes internally.
◈ A private variable `__balance` is internally stored as `_ClassName__balance`.

```python
python
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance

# Creating an object
account = BankAccount(5000)

# Accessing private attribute using name mangling (not recommended)
print(account._BankAccount__balance)  # ✅ Possible but not recommended
```

Output:

```
5000
```

⚠ **Name mangling is not meant to be used directly.** It's a trick to prevent accidental access, not a security feature.

Chapter 4: Inheritance – Code Reusability and Hierarchy

In this chapter, we will explore **Inheritance**, a powerful OOP concept that allows us to create new classes from existing ones. We will learn how to reuse code efficiently, understand different types of inheritance, and see real-world applications.

4.1 What is Inheritance?

Inheritance allows a new class (child) to **acquire** the properties and methods of an existing class (parent).

Why is Inheritance Important?

✅ **Code reusability** – No need to rewrite the same code multiple times.
✅ **Hierarchy & Structure** – Helps organize classes in a logical way.
✅ **Extensibility** – Easily modify or extend existing functionality.

4.2 Basic Syntax of Inheritance

To **inherit** a class in Python, we pass the parent class inside the parentheses of the child class:

```python
python
class Parent:
    # Constructor
    def __init__(self, name):
        self.name = name
```

```
    def show(self):
        print(f"Name: {self.name}")

# Child class inherits Parent
class Child(Parent):
    pass  # No additional attributes or methods

# Creating an object of Child
c = Child("Alice")
c.show()  # ✅ Inherited method from Parent
```

Output:

```
Name: Alice
```

◈ Even though Child has no methods, it **inherits** show() from Parent.
◈ The **child class** gets all the attributes and methods of the **parent class**.

4.3 Types of Inheritance in Python

Python supports **5 types of inheritance**:

1️⃣ **Single Inheritance** – One parent, one child.
2️⃣ **Multiple Inheritance** – One child, multiple parents.
3️⃣ **Multilevel Inheritance** – A child inherits from another child class.
4️⃣ **Hierarchical Inheritance** – One parent, multiple children.
5️⃣ **Hybrid Inheritance** – A combination of multiple types.

Let's explore them one by one!

4.4 Single Inheritance (One Parent, One Child)

Single inheritance means a **child class inherits** from a **single parent class**.

```
python
class Animal:
    def __init__(self, species):
        self.species = species

    def speak(self):
        print("Animal makes a sound")

# Dog class inherits Animal
class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__("Dog")  # Call parent constructor
        self.name = name
        self.breed = breed

    def bark(self):
        print(f"{self.name} is barking!")
```

```
# Creating an object of Dog
dog1 = Dog("Buddy", "Golden Retriever")
dog1.speak()  # ✅ Inherited from Animal
dog1.bark()   # ✅ Defined in Dog
```

Output:

```
Animal makes a sound
Buddy is barking!
```

🔷 `Dog` gets the **speak()** method from `Animal`.
🔷 It also has its own method **bark()**.

4.5 Multiple Inheritance (One Child, Multiple Parents)

A class can **inherit from multiple parents** at the same time.

```
python
class Father:
    def show_father(self):
        print("Father's property")

class Mother:
    def show_mother(self):
        print("Mother's property")

# Child inherits from both Father and Mother
class Child(Father, Mother):
    def show_child(self):
        print("Child's own property")

# Creating an object of Child
c = Child()
c.show_father()  # ✅ Inherited from Father
c.show_mother()  # ✅ Inherited from Mother
c.show_child()   # ✅ Defined in Child
```

Output:

```
Father's property
Mother's property
Child's own property
```

🔷 `Child` gets properties from **both Father and Mother**.
🔷 Python follows **Method Resolution Order (MRO)** when multiple parents have the same method.

4.6 Multilevel Inheritance (Grandparent → Parent → Child)

In **multilevel inheritance**, a child class inherits from another child class, forming a **chain**.

```
python
class Grandparent:
    def show_grandparent(self):
        print("Grandparent's property")

class Parent(Grandparent):
    def show_parent(self):
        print("Parent's property")

class Child(Parent):
    def show_child(self):
        print("Child's own property")


# Creating an object of Child
c = Child()
c.show_grandparent()   # ✅ Inherited from Grandparent
c.show_parent()        # ✅ Inherited from Parent
c.show_child()         # ✅ Defined in Child
```

Output:

```
Grandparent's property
Parent's property
Child's own property
```

◈ `Child` inherits from `Parent`, which **already inherits from `Grandparent`**.
◈ This forms a **hierarchical structure** of inheritance.

4.7 Hierarchical Inheritance (One Parent, Multiple Children)

In **hierarchical inheritance**, multiple child classes **inherit** from the **same parent class**.

```
python
class Vehicle:
    def show_vehicle(self):
        print("This is a vehicle")

class Car(Vehicle):
    def show_car(self):
        print("This is a car")

class Bike(Vehicle):
    def show_bike(self):
        print("This is a bike")

# Creating objects
car = Car()
```

```
bike = Bike()

car.show_vehicle()   # ✅ Inherited from Vehicle
car.show_car()       # ✅ Defined in Car

bike.show_vehicle()  # ✅ Inherited from Vehicle
bike.show_bike()     # ✅ Defined in Bike
```

Output:

```
This is a vehicle
This is a car
This is a vehicle
This is a bike
```

🔷 `Car` and `Bike` **both inherit** from `Vehicle`, but they have **different behaviors**.

4.8 Method Overriding in Inheritance

🔷 **Child classes can override parent class methods** by defining them again.

```
python
class Animal:
    def speak(self):
        print("Animal makes a sound")

class Dog(Animal):
    def speak(self):  # Overriding method
        print("Dog barks!")

# Creating an object
d = Dog()
d.speak()  # ✅ Calls the overridden method in Dog
```

Output:

```
Dog barks!
```

🔷 The `speak()` method in `Dog` **overrides** the `speak()` method in `Animal`.

4.9 The `super()` Function

🔷 `super()` **calls the parent class method** inside a child class.

```
python
class Parent:
    def show(self):
        print("This is the parent class")

class Child(Parent):
```

```
    def show(self):
        super().show()  # Call parent method
        print("This is the child class")

# Creating an object
c = Child()
c.show()
```

Output:

```
This is the parent class
This is the child class
```

Chapter 5: Polymorphism – Flexibility in OOP

In this chapter, we will explore **Polymorphism**, a key concept in Object-Oriented Programming that allows different classes to use the **same method name** but behave differently. We will learn about **method overloading, method overriding, and operator overloading** with real-world examples.

5.1 What is Polymorphism?

Polymorphism means "**one name, many forms**." It allows different objects to **respond to the same method differently** based on their class.

Why is Polymorphism Important?

✅ **Code flexibility** – The same function can work with different types of objects.
✅ **Extensibility** – New classes can reuse existing method names.
✅ **Reduces complexity** – No need to remember different function names for similar actions.

5.2 Types of Polymorphism in Python

Python supports **three types of polymorphism**:

1️⃣ **Method Overriding** – A child class redefines a method from the parent class.
2️⃣ **Method Overloading** – The same method can have different numbers of parameters.
3️⃣ **Operator Overloading** – Using built-in operators (`+`, `-`, `*`, etc.) for custom behavior.

Let's explore each of them with examples!

5.3 Method Overriding (Redefining Methods in a Child Class)

◈ **Method Overriding** occurs when a child class **redefines a method** from its parent class to give it a new behavior.

```
python
class Animal:
    def speak(self):
        print("Animal makes a sound")

class Dog(Animal):
    def speak(self):  # Overriding the parent method
        print("Dog barks!")

class Cat(Animal):
    def speak(self):  # Overriding the parent method
        print("Cat meows!")

# Creating objects
a = Animal()
d = Dog()
c = Cat()

# Calling the overridden method
a.speak()  # ✅ Uses Animal's method
d.speak()  # ✅ Uses Dog's overridden method
c.speak()  # ✅ Uses Cat's overridden method
```

Output:

```
Animal makes a sound
Dog barks!
Cat meows!
```

◈ The `speak()` method behaves **differently** for each class, even though they share the same method name.
◈ This is **runtime polymorphism** (the method is decided at runtime).

5.4 Method Overloading (Multiple Methods with the Same Name)

◈ **Python does not support method overloading directly** like Java or C++.
◈ However, we can achieve similar behavior using **default arguments** or `*args`.

Example 1: Using Default Arguments

```
python
class Calculator:
    def add(self, a, b=0, c=0):
        return a + b + c  # Works with 1, 2, or 3 arguments

calc = Calculator()
```

```
print(calc.add(5))        # ✅ 5 + 0 + 0 = 5
print(calc.add(5, 10))    # ✅ 5 + 10 + 0 = 15
print(calc.add(5, 10, 2)) # ✅ 5 + 10 + 2 = 17
```

Output:

```
5
15
17
```

Example 2: Using `*args` (Variable Arguments)

```
python
class MathOperations:
    def multiply(self, *nums):  # Accepts multiple numbers
        result = 1
        for num in nums:
            result *= num
        return result

math = MathOperations()
print(math.multiply(2))          # ✅ 2
print(math.multiply(2, 3))       # ✅ 2 * 3 = 6
print(math.multiply(2, 3, 4))    # ✅ 2 * 3 * 4 = 24
```

Output:

```
2
6
24
```

◈ `*args` allows **flexible arguments**, simulating **method overloading**.

5.5 Operator Overloading (Customizing Operators for Classes)

◈ **Operator overloading** allows us to **modify how built-in operators (+, -, *, etc.) work with objects.**
◈ This is done using **special methods (dunder methods)** like `__add__`, `__sub__`, etc.

Example 1: Overloading + for Custom Classes

```
python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):  # Overloading `+`
        return Point(self.x + other.x, self.y + other.y)
```

```
    def show(self):
        print(f"Point({self.x}, {self.y})")

# Creating points
p1 = Point(2, 3)
p2 = Point(4, 5)

# Adding two objects
p3 = p1 + p2  # ✅ Works because of __add__
p3.show()
```

Output:

```
Point(6, 8)
```

◈ The + operator now **adds two points** instead of numbers!

5.6 More Operator Overloading Examples

Operator	Overloaded Method	Example
+	__add__(self, other)	obj1 + obj2
-	__sub__(self, other)	obj1 - obj2
*	__mul__(self, other)	obj1 * obj2
/	__truediv__(self, other)	obj1 / obj2
==	__eq__(self, other)	obj1 == obj2
!=	__ne__(self, other)	obj1!= obj2
>	__gt__(self, other)	obj1 > obj2
<	__lt__(self, other)	obj1 < obj2

Example 2: Overloading * (Multiplication)

```
python
class Number:
    def __init__(self, value):
        self.value = value

    def __mul__(self, other):  # Overloading `*`
        return Number(self.value * other.value)

    def show(self):
        print(f"Value: {self.value}")

# Creating objects
num1 = Number(5)
num2 = Number(3)

# Multiplying objects
num3 = num1 * num2  # ✅ Uses __mul__
num3.show()
```

Output:

```
Value: 15
```

◈ The `*` operator now **multiplies custom objects** instead of numbers!

Chapter 6: Abstraction – Hiding Implementation Details

In this chapter, we will explore **Abstraction**, a crucial concept in Object-Oriented Programming (OOP) that focuses on **hiding the internal details** of how something works while exposing only the necessary functionality. We will learn how to achieve abstraction using **abstract classes and methods** in Python, along with real-world examples and best practices.

6.1 What is Abstraction?

◈ **Abstraction** is the process of **hiding implementation details** and **only showing relevant features** to the user.
◈ It helps in reducing complexity and making code more maintainable.

Example: Car Abstraction

When you drive a car:
✅ You **press the accelerator**, but you **don't need to know** how fuel combustion works.
✅ You **turn the steering wheel**, but you **don't need to understand** the internal mechanics.

Similarly, in programming, **abstraction hides the details** and only provides **necessary functionalities**.

Key Benefits of Abstraction

✅ **Simplifies complex systems** by breaking them into smaller parts.
✅ **Improves security** by preventing access to unnecessary details.
✅ **Enhances flexibility** – We can change implementations without affecting users.

6.2 Abstraction in Python using Abstract Classes

◈ **Abstract Classes** are classes that **cannot be instantiated directly**.
◈ They **define a blueprint** for other classes by including **abstract methods** (methods without implementation).

How to Create an Abstract Class in Python?

We use the **`ABC` (Abstract Base Class) module** from the `abc` library.

Syntax: Abstract Class and Abstract Methods

```
python
from abc import ABC, abstractmethod

class Vehicle(ABC):  # Abstract Class
    @abstractmethod
    def start(self):
        pass  # Abstract Method (no implementation)

    @abstractmethod
    def stop(self):
        pass  # Abstract Method

# Concrete Class (Child)
class Car(Vehicle):
    def start(self):
        print("Car engine started!")

    def stop(self):
        print("Car engine stopped!")

# Creating an object
# v = Vehicle()  ❌ ERROR! Cannot instantiate abstract class

c = Car()  # ✅ Allowed
c.start()
c.stop()
```

Output:

```
Car engine started!
Car engine stopped!
```

◈ **Abstract methods (`start()` and `stop()`)** are **defined but not implemented** in the `Vehicle` class.
◈ The `Car` class **implements those methods**, so it can be instantiated.
◈ If a child class **fails to implement** all abstract methods, it will cause an error!

6.3 Partial Abstraction (Abstract + Concrete Methods)

◈ Abstract classes **can have both** abstract methods and concrete methods (with implementations).

```
python
from abc import ABC, abstractmethod

class Animal(ABC):
```

```
    @abstractmethod
    def sound(self):  # Abstract Method
        pass

    def sleep(self):  # Concrete Method
        print("Animals need sleep.")

class Dog(Animal):
    def sound(self):
        print("Dog barks!")

# Creating an object
d = Dog()
d.sound()  # ✅ Overridden method
d.sleep()  # ✅ Inherited concrete method
```

Output:

```
Dog barks!
Animals need sleep.
```

◈ The `Dog` class **inherits both** abstract (`sound()`) and concrete (`sleep()`) methods from `Animal`.

◈ **Concrete methods** are useful when all child classes need the same behavior.

6.4 Real-World Example: Payment System

Consider an online payment system where different **payment methods** (Credit Card, PayPal, etc.) share a common structure but have different implementations.

```
python
from abc import ABC, abstractmethod

class Payment(ABC):
    @abstractmethod
    def pay(self, amount):
        pass

class CreditCardPayment(Payment):
    def pay(self, amount):
        print(f"Paid ${amount} using Credit Card.")

class PayPalPayment(Payment):
    def pay(self, amount):
        print(f"Paid ${amount} using PayPal.")

# Creating objects
payment1 = CreditCardPayment()
payment1.pay(100)  # ✅ Paid using Credit Card

payment2 = PayPalPayment()
payment2.pay(50)  # ✅ Paid using PayPal
```

Output:

```
Paid $100 using Credit Card.
Paid $50 using PayPal.
```

◈ `Payment is an abstract class that` **`defines a blueprint`** `(pay()).`
◈ `Each subclass` **`implements`** `pay() in its own way.`

Chapter 7: Encapsulation – Data Protection in OOP

Encapsulation is one of the four fundamental principles of Object-Oriented Programming (OOP). It helps in **hiding data** and **restricting direct access** to certain variables and methods, ensuring better **security and maintainability**.

7.1 What is Encapsulation?

◈ **Encapsulation** is the technique of **wrapping data (variables) and methods together** into a single unit (class).
◈ It **restricts direct access** to some data, allowing controlled access through methods.
◈ This ensures **data security and prevents accidental modification**.

Example: Bank Account System

When you use a bank account, you **don't directly modify your balance**. Instead, you use **methods like `deposit()` and `withdraw()`**.

☑ **Good practice: Hiding balance** and allowing access through methods.
✗ **Bad practice:** Directly modifying the balance (`account.balance = -500`).

7.2 Implementing Encapsulation in Python

Python uses **access modifiers** to control data access:

Modifier	Syntax	Access
Public	`self.variable`	Accessible from anywhere
Protected	`self._variable`	Suggests restricted access (used within class and subclasses)
Private	`self.__variable`	Cannot be accessed directly from outside the class

7.3 Private and Protected Attributes

◈ **Protected Attributes** (`_attribute`): Not enforced by Python but suggests limited access.
◈ **Private Attributes** (`__attribute`): Cannot be accessed directly from outside the class.

Example: Using Private and Protected Attributes

```
python
class BankAccount:
    def __init__(self, account_holder, balance):
        self.account_holder = account_holder  # Public
        self._interest_rate = 5  # Protected
        self.__balance = balance  # Private

    def deposit(self, amount):
        self.__balance += amount
        print(f"${amount} deposited. New balance: ${self.__balance}")

    def withdraw(self, amount):
        if amount > self.__balance:
            print("Insufficient balance!")
        else:
            self.__balance -= amount
            print(f"${amount} withdrawn. Remaining balance:
${self.__balance}")

    def get_balance(self):
        return self.__balance  # Controlled access to private attribute

# Creating an account
account = BankAccount("John Doe", 1000)

# Accessing public attribute
print(account.account_holder)  # ✅ Allowed

# Accessing protected attribute (Not recommended but possible)
print(account._interest_rate)  # ✅ Allowed but discouraged

# Accessing private attribute directly (Not allowed)
# print(account.__balance)  ❌ Error

# Correct way to access private attribute
print(account.get_balance())  # ✅ Allowed through method
```

Output:

```
John Doe
5
1000
```

◈ `__balance` is **private**, so it cannot be accessed directly.
◈ `get_balance()` **allows controlled access** to the private attribute.

7.4 Getters and Setters (Controlled Access to Data)

Getters and **Setters** allow **safe access** to private attributes:

```
python
class Student:
    def __init__(self, name, age):
        self.__name = name   # Private
        self.__age = age     # Private

    def get_age(self):  # Getter
        return self.__age

    def set_age(self, new_age):  # Setter
        if new_age > 0:
            self.__age = new_age
        else:
            print("Invalid age!")

# Creating a student
s = Student("Alice", 20)

# Accessing age safely
print(s.get_age())  # ✅ 20

# Updating age safely
s.set_age(25)  # ✅ Allowed
print(s.get_age())  # ✅ 25

# Trying to set an invalid age
s.set_age(-5)  # ❌ Invalid age!
```

◈ **Getters retrieve private data**, and **setters update private data safely**.

7.5 Property Decorators (`@property`)

Python provides a more **elegant way** to use getters and setters with `@property`.

```
python
class Employee:
    def __init__(self, name, salary):
        self.__name = name
        self.__salary = salary

    @property
    def salary(self):  # Getter
        return self.__salary

    @salary.setter
```

```
    def salary(self, new_salary):  # Setter
        if new_salary > 0:
            self.__salary = new_salary
        else:
            print("Invalid salary!")

# Creating an employee
e = Employee("David", 5000)

# Accessing and modifying salary using @property
print(e.salary)  # ✅ 5000
e.salary = 6000  # ✅ Allowed
print(e.salary)  # ✅ 6000
e.salary = -100  # ❌ Invalid salary!
```

🔷 Using `@property`, we can **access methods like attributes** (`e.salary` instead of `e.get_salary()`).

7.6 Real-World Example: Encapsulated Car System

```python
class Car:
    def __init__(self, model, fuel_capacity):
        self.model = model  # Public
        self.__fuel = fuel_capacity  # Private

    def refuel(self, amount):
        if amount > 0:
            self.__fuel += amount
            print(f"Refueled {amount} liters. Current fuel: {self.__fuel}L")
        else:
            print("Invalid fuel amount!")

    def get_fuel(self):
        return self.__fuel

# Creating a car object
car = Car("Toyota Corolla", 40)

# Trying to access private fuel directly (Not allowed)
# print(car.__fuel)  ❌ Error

# Using the method to get fuel
print(car.get_fuel())  # ✅ 40

# Refueling safely
car.refuel(10)  # ✅ 50L total
car.refuel(-5)  # ❌ Invalid amount!
```

Chapter 8: Class Relationships – How Objects Interact

In Object-Oriented Programming (OOP), **classes don't exist in isolation**—they interact with each other in different ways. Understanding **class relationships** is crucial for designing real-world applications efficiently.

8.1 What Are Class Relationships?

Class relationships define **how objects of different classes interact**. The main types of relationships are:

1 **Association** – A general relationship where objects use each other.
2 **Aggregation** – A "whole-part" relationship where parts can exist independently.
3 **Composition** – A stronger "whole-part" relationship where parts depend on the whole.
4 **Inheritance** – A "parent-child" relationship where a subclass inherits from a superclass.

8.2 Association (General Relationship Between Classes)

◈ **Definition**: Association is when **two classes are related but neither "owns" the other**.
◈ **Example**: A `Driver` and a `Car` are associated because a driver can drive a car, but they are independent.

Example: Association

```
python
class Driver:
    def __init__(self, name):
        self.name = name

    def drive(self, car):
        print(f"{self.name} is driving the {car.model}.")

class Car:
    def __init__(self, model):
        self.model = model

# Creating objects
driver = Driver("John")
car = Car("Toyota Corolla")

# Establishing an association
driver.drive(car)
```

Output:

```
John is driving the Toyota Corolla.
```

◈ The `Driver` and `Car` classes **are independent but interact** through the `drive()` method.

8.3 Aggregation (Has-A Relationship, Loose Coupling)

◈ **Definition**: Aggregation is a **"has-a"** relationship where one class **contains another as a part**, but the contained object can exist independently.
◈ **Example**: A `Library` **has** multiple `Books`, but `Books` **can exist without the Library**.

Example: Aggregation

```
python
class Book:
    def __init__(self, title):
        self.title = title

class Library:
    def __init__(self):
        self.books = []  # List to store book objects

    def add_book(self, book):
        self.books.append(book)

    def show_books(self):
        for book in self.books:
            print(f"Book: {book.title}")

# Creating books
book1 = Book("Python for Beginners")
book2 = Book("OOP in Python")

# Creating library and adding books
library = Library()
library.add_book(book1)
library.add_book(book2)

# Displaying books
library.show_books()
```

Output:

```
Book: Python for Beginners
Book: OOP in Python
```

◈ The `Library` **has books**, but `Book` objects **exist independently**.

8.4 Composition (Has-A Relationship, Strong Coupling)

◈ **Definition**: Composition is also a **"has-a"** relationship, but here, the **part cannot exist without the whole**.
◈ **Example**: A `Car` has an `Engine`, but an `Engine` **cannot function without the Car**.

Example: Composition

```python
class Engine:
    def __init__(self, horsepower):
        self.horsepower = horsepower

class Car:
    def __init__(self, model, horsepower):
        self.model = model
        self.engine = Engine(horsepower)  # Composition: Car owns Engine

    def show_details(self):
        print(f"Car: {self.model}, Engine: {self.engine.horsepower} HP")

# Creating a car
car = Car("Tesla Model 3", 400)
car.show_details()
```

Output:

```
Car: Tesla Model 3, Engine: 400 HP
```

◈ The `Engine` **is part of** the `Car`, and **it cannot exist independently**.

8.5 Inheritance (Is-A Relationship)

◈ **Definition**: Inheritance represents an **"is-a"** relationship where a **subclass derives from a superclass**.
◈ **Example**: A `Dog` **is a** type of `Animal`.

Example: Inheritance

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print("Animal makes a sound.")

class Dog(Animal):  # Dog inherits from Animal
    def speak(self):
        print(f"{self.name} barks!")


# Creating objects
dog = Dog("Buddy")
dog.speak()
```

Output:

```
Buddy barks!
```

◈ The `Dog` class **inherits** from `Animal` but **overrides** the `speak()` method.

8.6 Key Differences Between Aggregation, Composition, and Inheritance

Concept	Relationship Type	Can Exist Independently?	Example
Association	Uses another class	Yes	Driver & Car
Aggregation	Has-a (Loose)	Yes	Library & Book
Composition	Has-a (Strong)	No	Car & Engine
Inheritance	Is-a	No (Child depends on Parent)	Dog & Animal

8.7 Real-World Example: School Management System

```
python
class Student:
    def __init__(self, name, student_id):
        self.name = name
        self.student_id = student_id

class Course:
    def __init__(self, course_name):
        self.course_name = course_name
        self.students = []  # Aggregation: Students can exist independently

    def add_student(self, student):
        self.students.append(student)

    def show_students(self):
        print(f"Students in {self.course_name}:")
        for student in self.students:
            print(f"- {student.name} (ID: {student.student_id})")

# Creating students
student1 = Student("Alice", 101)
student2 = Student("Bob", 102)

# Creating course
course = Course("Python Programming")
course.add_student(student1)
course.add_student(student2)


# Displaying students
course.show_students()
```

Output:

```
Students in Python Programming:
- Alice (ID: 101)
- Bob (ID: 102)
```

◈ **Aggregation**: `Course` has `Students`, but `Students` **can exist without a Course**.

Chapter 9: Design Patterns in OOP

Design patterns are **proven solutions** to **common programming problems** in software development. They help in writing **efficient, reusable, and maintainable code** by providing structured approaches to solving design challenges.

9.1 What Are Design Patterns?

◈ **Definition:** A design pattern is a **general repeatable solution** to a common problem in software design.
◈ **Purpose:** They provide **best practices** for structuring code in an efficient and scalable way.
◈ **Types of Design Patterns:**

1 **Creational Patterns** – Focus on object creation mechanisms.
2 **Structural Patterns** – Focus on organizing objects and classes.
3 **Behavioral Patterns** – Focus on communication between objects.

9.2 Creational Design Patterns

9.2.1 Singleton Pattern (Ensuring a Single Instance)

◈ **Use Case:** When **only one instance** of a class should exist (e.g., Database Connection, Configuration Manager).
◈ **Implementation:**

```
python
class Singleton:
    _instance = None  # Class attribute to store the single instance

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(Singleton, cls).__new__(cls)
        return cls._instance

# Creating objects
obj1 = Singleton()
obj2 = Singleton()

print(obj1 is obj2)  # ✅ True (Both refer to the same instance)
```

◈ **Both `obj1` and `obj2` point to the same object, ensuring a single instance.**

9.2.2 Factory Method (Creating Objects Without Specifying Exact Class)

◈ **Use Case:** When we need to create **objects dynamically** without exposing the exact class.
◈ **Implementation:**

```
python
class Dog:
    def speak(self):
        return "Woof!"

class Cat:
    def speak(self):
        return "Meow!"

class AnimalFactory:
    @staticmethod
    def get_animal(animal_type):
        if animal_type == "dog":
            return Dog()
        elif animal_type == "cat":
            return Cat()
        else:
            return None

# Using the factory
animal = AnimalFactory.get_animal("dog")
print(animal.speak())  # ✅ Woof!
```

◈ **The factory method provides flexibility in object creation.**

9.3 Structural Design Patterns

9.3.1 Adapter Pattern (Converting One Interface to Another)

◈ **Use Case:** When two classes **cannot work together** due to different interfaces.
◈ **Example:** A `USB` device that needs to be plugged into an `HDMI` port.
◈ **Implementation:**

```
python
class EuropeanPlug:
    def power(self):
        return "Powering device with European plug"

class Adapter:
    def __init__(self, plug):
        self.plug = plug
```

```
    def power(self):
        return self.plug.power() + " via Adapter"

# Using the adapter
plug = EuropeanPlug()
adapter = Adapter(plug)
print(adapter.power())  # ✅ Powering device with European plug via Adapter
```

◈ **Adapters act as a bridge between incompatible interfaces.**

9.3.2 Decorator Pattern (Adding Functionality Without Modifying Original Class)

◈ **Use Case:** When we need to **extend functionality** of an object **without modifying** its class.
◈ **Example:** Adding logging to a function dynamically.
◈ **Implementation:**

```
python
def decorator(func):
    def wrapper():
        print("Logging before function call...")
        func()
        print("Logging after function call...")
    return wrapper

@decorator
def say_hello():
    print("Hello, World!")

say_hello()
```

Output:

```
Logging before function call...
Hello, World!
Logging after function call...
```

◈ **Decorator pattern allows adding features dynamically.**

9.4 Behavioral Design Patterns

9.4.1 Observer Pattern (Notifying Multiple Objects of a Change)

◈ **Use Case:** When multiple objects **need to be notified** if some data changes (e.g., Notification system).
◈ **Implementation:**

```python
python
class Observer:
    def update(self, message):
        pass

class User(Observer):
    def __init__(self, name):
        self.name = name

    def update(self, message):
        print(f"{self.name} received notification: {message}")

class NotificationSystem:
    def __init__(self):
        self.subscribers = []

    def subscribe(self, user):
        self.subscribers.append(user)

    def notify_all(self, message):
        for subscriber in self.subscribers:
            subscriber.update(message)

# Using Observer Pattern
notifier = NotificationSystem()
user1 = User("Alice")
user2 = User("Bob")

notifier.subscribe(user1)
notifier.subscribe(user2)

notifier.notify_all("New update available!")
```

Output:

```
Alice received notification: New update available!
Bob received notification: New update available!
```

◈ **The Observer Pattern is widely used in event-driven systems.**

9.4.2 Strategy Pattern (Selecting Algorithms Dynamically at Runtime)

◈ **Use Case:** When different strategies (algorithms) need to be swapped dynamically.
◈ **Example:** A payment system that supports multiple payment methods.
◈ **Implementation:**

```python
python
class PaymentStrategy:
    def pay(self, amount):
        pass
```

```python
class CreditCardPayment(PaymentStrategy):
    def pay(self, amount):
        print(f"Paid ${amount} using Credit Card.")

class PayPalPayment(PaymentStrategy):
    def pay(self, amount):
        print(f"Paid ${amount} using PayPal.")

class PaymentContext:
    def __init__(self, strategy):
        self.strategy = strategy

    def execute_payment(self, amount):
        self.strategy.pay(amount)

# Using different payment methods
credit_card = PaymentContext(CreditCardPayment())
credit_card.execute_payment(100)

paypal = PaymentContext(PayPalPayment())
paypal.execute_payment(200)
```

Output:

```
Paid $100 using Credit Card.
Paid $200 using PayPal.
```

◈ **The strategy pattern allows flexible selection of different behaviors at runtime.**

9.5 Summary of Design Patterns

Pattern Name	Type	Use Case
Singleton	Creational	Ensures only **one instance** of a class exists.
Factory Method	Creational	Creates objects **without specifying the exact class**.
Adapter	Structural	Converts **one interface to another**.
Decorator	Structural	**Extends functionality** dynamically.
Observer	Behavioral	**Notifies multiple objects** of a change.
Strategy	Behavioral	**Switches algorithms dynamically** at runtime.

Chapter 10: File Handling in Object-Oriented Programming (OOP)

File handling is an essential part of software development, allowing programs to **store, retrieve, and manipulate data** persistently. In OOP, we use **classes and objects** to manage file operations efficiently.

10.1 Why Use File Handling in OOP?

◈ **Persistence** – Store data permanently beyond program execution.
◈ **Modular Design** – Keep data separate from logic, improving maintainability.
◈ **Data Management** – Read, write, update, and delete files dynamically.

10.2 Basic File Operations in Python

Python provides built-in functions for file handling:

Operation	Method	Description
Read	`open(filename, "r")`	Read a file's content
Write	`open(filename, "w")`	Write new content (overwrites existing)
Append	`open(filename, "a")`	Add content to the end of a file
Binary Read/Write	`open(filename, "rb" or "wb")`	Read/write non-text files (e.g., images, PDFs)

10.3 File Handling Using OOP – Creating a File Manager Class

A **FileManager** class can encapsulate all file-related operations.

Example: Basic File Handling Class

```
python
class FileManager:
    def __init__(self, filename):
        self.filename = filename

    def write_file(self, content):
        with open(self.filename, "w") as file:
            file.write(content)
        print("File written successfully.")

    def read_file(self):
```

```
        with open(self.filename, "r") as file:
            return file.read()

    def append_file(self, content):
        with open(self.filename, "a") as file:
            file.write(content)
        print("Content appended successfully.")

# Usage
file_manager = FileManager("data.txt")

# Writing to file
file_manager.write_file("Hello, this is a test file.")

# Reading from file
print(file_manager.read_file())

# Appending to file
file_manager.append_file("\nAppending new content.")

# Reading again
print(file_manager.read_file())
```

Output:

```
File written successfully.
Hello, this is a test file.
Content appended successfully.
Hello, this is a test file.
Appending new content.
```

◈ **Encapsulating file operations** into a class makes it reusable and maintainable.

10.4 Handling Exceptions in File Operations

◈ **Files may not exist, or errors may occur while reading/writing.**
◈ We should **handle exceptions** to prevent crashes.

Example: Safe File Handling with Exception Handling

```
python
class SafeFileManager:
    def __init__(self, filename):
        self.filename = filename

    def read_file(self):
        try:
            with open(self.filename, "r") as file:
                return file.read()
        except FileNotFoundError:
            return "Error: File not found!"
        except Exception as e:
            return f"Error: {e}"
```

```
# Usage
safe_manager = SafeFileManager("nonexistent.txt")
print(safe_manager.read_file())  # ✅ Error: File not found!
```

◈ **Handling errors gracefully** prevents unexpected crashes.

10.5 Working with CSV Files in OOP

◈ **CSV (Comma-Separated Values)** files store tabular data in a structured format.
◈ Python's `csv` module allows us to read and write CSV files efficiently.

Example: Managing CSV Files with OOP

```
python
import csv

class CSVManager:
    def __init__(self, filename):
        self.filename = filename

    def write_csv(self, data):
        with open(self.filename, "w", newline="") as file:
            writer = csv.writer(file)
            writer.writerows(data)
        print("CSV file written successfully.")

    def read_csv(self):
        with open(self.filename, "r") as file:
            reader = csv.reader(file)
            return [row for row in reader]

# Usage
csv_manager = CSVManager("students.csv")

# Writing data to CSV
csv_manager.write_csv([["Name", "Age"], ["Alice", 22], ["Bob", 24]])

# Reading data from CSV
print(csv_manager.read_csv())
```

Output:

```
CSV file written successfully.
[['Name', 'Age'], ['Alice', '22'], ['Bob', '24']]
```

◈ **The CSVManager class provides reusable methods for handling CSV files.**

10.6 Working with JSON Files in OOP

◈ **JSON (JavaScript Object Notation)** is a lightweight format for storing structured data.
◈ Python's `json` module enables reading and writing JSON files.

Example: Managing JSON Files with OOP

```python
import json

class JSONManager:
    def __init__(self, filename):
        self.filename = filename

    def write_json(self, data):
        with open(self.filename, "w") as file:
            json.dump(data, file, indent=4)
        print("JSON file written successfully.")

    def read_json(self):
        with open(self.filename, "r") as file:
            return json.load(file)

# Usage
json_manager = JSONManager("data.json")

# Writing JSON
json_manager.write_json({"name": "Alice", "age": 22})

# Reading JSON
print(json_manager.read_json())
```

Output:

```
JSON file written successfully.
{'name': 'Alice', 'age': 22}
```

◈ **The JSONManager class handles structured data efficiently.**

10.7 File Handling Best Practices

✅ **Use `with open()`** – Ensures files close automatically after use.
✅ **Handle exceptions** – Prevents errors from crashing the program.
✅ **Use structured data formats (CSV/JSON)** – Easier to read and write.
✅ **Keep file operations encapsulated in classes** – Improves code maintainability.

Chapter 11: Database Integration in Object-Oriented Programming (OOP)

In real-world applications, data is often stored in **databases** instead of files. This chapter covers how to integrate **databases with Python using OOP principles** to manage data efficiently.

11.1 Why Use a Database Instead of Files?

Feature	File System	Database
Scalability	Difficult to manage large data	Handles large datasets efficiently
Security	Limited access control	Strong authentication and encryption
Data Integrity	Requires manual handling	Ensures consistency and relationships
Performance	Slower search operations	Faster queries using indexing
Multi-user Access	Hard to manage	Supports multiple users concurrently

Conclusion: Databases are the best choice for structured, secure, and scalable data management.

11.2 Connecting Python with SQLite (Lightweight Database)

◈ **SQLite** is a lightweight database that comes **pre-installed** with Python.
◈ Python's `sqlite3` module allows us to interact with an SQLite database easily.

Example: Creating a Database Connection

```
python
import sqlite3

class DatabaseManager:
    def __init__(self, db_name):
        self.connection = sqlite3.connect(db_name)
        self.cursor = self.connection.cursor()

    def close_connection(self):
        self.connection.close()

# Usage
db = DatabaseManager("example.db")
print("Database connected successfully.")
db.close_connection()
```

◈ **This class handles database connections dynamically.**

11.3 Creating Tables in OOP

To store structured data, we need **tables** inside the database.

Example: Creating a Users Table

```
python
class UserDatabase(DatabaseManager):
    def create_table(self):
        self.cursor.execute("""
            CREATE TABLE IF NOT EXISTS users (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                name TEXT,
                age INTEGER
            )
        """)
        self.connection.commit()
        print("Table created successfully.")

# Usage
user_db = UserDatabase("users.db")
user_db.create_table()
user_db.close_connection()
```

◈ **`CREATE TABLE IF NOT EXISTS` ensures we don't create duplicate tables.**

11.4 Inserting Data into the Database

Example: Inserting User Data

```
python
class UserDatabase(DatabaseManager):
    def insert_user(self, name, age):
        self.cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)",
(name, age))
        self.connection.commit()
        print("User added successfully.")

# Usage
user_db = UserDatabase("users.db")
user_db.insert_user("Alice", 25)
user_db.insert_user("Bob", 30)
user_db.close_connection()
```

◈ **Using `?` placeholders prevents SQL injection attacks.**

11.5 Fetching Data from the Database

Example: Retrieving All Users

python
```
class UserDatabase(DatabaseManager):
    def fetch_users(self):
        self.cursor.execute("SELECT * FROM users")
        return self.cursor.fetchall()

# Usage
user_db = UserDatabase("users.db")
users = user_db.fetch_users()
for user in users:
    print(user)  # (1, 'Alice', 25), (2, 'Bob', 30)
user_db.close_connection()
```

◈ **Data is returned as a list of tuples, representing database rows.**

11.6 Updating Data in the Database

Example: Updating a User's Age

python
```
class UserDatabase(DatabaseManager):
    def update_user(self, user_id, new_age):
        self.cursor.execute("UPDATE users SET age = ? WHERE id = ?",
(new_age, user_id))
        self.connection.commit()
        print("User updated successfully.")

# Usage
user_db = UserDatabase("users.db")
user_db.update_user(1, 28)  # Updates Alice's age to 28
user_db.close_connection()
```

◈ **Always use parameterized queries (`?`) to prevent security vulnerabilities.**

11.7 Deleting Data from the Database

Example: Deleting a User

python
```
class UserDatabase(DatabaseManager):
    def delete_user(self, user_id):
        self.cursor.execute("DELETE FROM users WHERE id = ?", (user_id,))
        self.connection.commit()
        print("User deleted successfully.")

# Usage
user_db = UserDatabase("users.db")
user_db.delete_user(2)  # Deletes Bob
```

```
user_db.close_connection()
```

◈ **`DELETE FROM` removes specific rows, while `DROP TABLE` removes the entire table.**

11.8 Object-Relational Mapping (ORM) with SQLAlchemy

◈ **SQLAlchemy** is a powerful library for handling databases in an **object-oriented way.**
◈ Instead of writing raw SQL, we define **Python classes as database tables.**

Example: Defining a Table Using SQLAlchemy

```
python
from sqlalchemy import create_engine, Column, Integer, String
from sqlalchemy.orm import declarative_base, sessionmaker

Base = declarative_base()

class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    age = Column(Integer)

# Creating the database
engine = create_engine("sqlite:///orm_users.db")
Base.metadata.create_all(engine)
Session = sessionmaker(bind=engine)
session = Session()

# Adding data using ORM
new_user = User(name="Charlie", age=35)
session.add(new_user)
session.commit()

# Fetching data using ORM
users = session.query(User).all()
for user in users:
    print(user.name, user.age)  # Charlie 35

session.close()
```

◈ **SQLAlchemy makes database operations more Pythonic and maintainable.**

11.9 Database Best Practices

☑ **Use parameterized queries (`?`)** – Prevents SQL injection attacks.
☑ **Close connections (`.close()`)** – Prevents memory leaks.

✅ **Use ORM (SQLAlchemy)** – Makes code more maintainable.
✅ **Implement proper error handling** – Catch database errors.

Chapter 12: Advanced Object-Oriented Programming (OOP) Concepts in Python

In this chapter, we will explore advanced OOP concepts, including **metaclasses, multiple inheritance, method resolution order (MRO), abstract base classes (ABCs), decorators, and design patterns**. These concepts will help you **write more efficient, flexible, and scalable object-oriented code**.

12.1 Metaclasses in Python

◈ **A metaclass is a class that defines how other classes behave.**
◈ In Python, every class is an **instance of a metaclass**, typically `type`.
◈ Metaclasses allow you to **modify class creation dynamically**.

Example: Custom Metaclass

```
CopyEdit
class Meta(type):
    def __new__(cls, name, bases, class_dict):
        print(f"Creating class: {name}")
        return super().__new__(cls, name, bases, class_dict)

class MyClass(metaclass=Meta):
    def hello(self):
        return "Hello from MyClass!"

# Output:
# Creating class: MyClass
```

◈ **Metaclasses are useful for enforcing coding standards, logging, and modifying behavior at class creation.**

12.2 Multiple Inheritance & The Diamond Problem

◈ **Multiple inheritance allows a class to inherit from multiple parent classes.**
◈ **The Diamond Problem** occurs when a class inherits from two classes that share a common ancestor.

Example: The Diamond Problem

```
python
class A:
    def show(self):
        print("A")

class B(A):
    def show(self):
        print("B")

class C(A):
    def show(self):
        print("C")

class D(B, C):  # Multiple Inheritance
    pass

d = D()
d.show()  # Output: B (follows MRO)
```

◈ Python **uses Method Resolution Order (MRO)** to resolve conflicts using the **C3 linearization algorithm**.

📌 **Check MRO using**:

```
python
print(D.mro())
```

12.3 Abstract Base Classes (ABCs) and Interfaces

◈ **Abstract Base Classes (ABCs)** enforce certain methods in child classes.
◈ ABCs act like **interfaces**, ensuring subclasses implement required methods.

Example: Using ABC in Python

```
python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass

class Dog(Animal):
```

```
    def sound(self):
        return "Bark"

class Cat(Animal):
    def sound(self):
        return "Meow"

# dog = Animal()  # ✕ Error: Cannot instantiate abstract class
dog = Dog()
print(dog.sound())  # Output: Bark
```

◈ **Use ABCs to define common structures for classes without allowing direct instantiation.**

12.4 Method Resolution Order (MRO) & The `super()` Function

◈ **MRO defines the order in which methods are inherited in multiple inheritance.**
◈ **Python follows the C3 linearization algorithm** (depth-first, left-to-right).

Example: Using `super()` in MRO

```
python
class A:
    def show(self):
        print("A")

class B(A):
    def show(self):
        super().show()  # Calls A's show() method
        print("B")

class C(A):
    def show(self):
        super().show()
        print("C")

class D(B, C):
    def show(self):
        super().show()
        print("D")

d = D()
d.show()
```

MRO for Class `D`

```
D → B → C → A
```

◈ **Use `super()` to avoid redundancy and ensure correct method calls in multiple inheritance.**

12.5 Class Decorators and Static Methods in OOP

◈ **Class decorators modify behavior at runtime without changing the original class.**
◈ **Static methods (`@staticmethod`) don't access instance variables and work like utility functions.**

Example: Class Decorators & Static Methods

```
python
def log_methods(cls):
    class NewClass(cls):
        def __getattribute__(self, name):
            print(f"Calling {name} method")
            return super().__getattribute__(name)
    return NewClass

@log_methods
class MathOperations:
    @staticmethod
    def add(a, b):
        return a + b

math_obj = MathOperations()
print(math_obj.add(5, 3))
# Output: Calling add method
#         8
```

◈ **Decorators modify behavior dynamically while keeping the code clean.**

12.6 Design Patterns in OOP

◈ **Design patterns provide reusable solutions to common software design problems.**
◈ We will cover **Singleton, Factory, and Observer patterns**.

1 Singleton Pattern (Ensures only one instance of a class exists)

```
python
class Singleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

s1 = Singleton()
s2 = Singleton()
print(s1 is s2)  # Output: True (Same instance)
```

2 Factory Pattern (Creates objects without specifying the exact class)

```python
class Animal:
    def make_sound(self):
        pass

class Dog(Animal):
    def make_sound(self):
        return "Bark"

class Cat(Animal):
    def make_sound(self):
        return "Meow"

class AnimalFactory:
    @staticmethod
    def create_animal(animal_type):
        if animal_type == "dog":
            return Dog()
        elif animal_type == "cat":
            return Cat()
        else:
            return None

animal = AnimalFactory.create_animal("dog")
print(animal.make_sound())  # Output: Bark
```

---

3 Observer Pattern (Notifies multiple objects when a change occurs)

```python
class Observer:
    def update(self, message):
        pass

class ConcreteObserver(Observer):
    def update(self, message):
        print(f"Received notification: {message}")

class Subject:
    def __init__(self):
        self.observers = []

    def add_observer(self, observer):
        self.observers.append(observer)

    def notify_observers(self, message):
        for observer in self.observers:
            observer.update(message)

# Usage
subject = Subject()
obs1 = ConcreteObserver()
obs2 = ConcreteObserver()

subject.add_observer(obs1)
subject.add_observer(obs2)
```

```
subject.notify_observers("New Update Available!")
```

12.7 Best Practices in Advanced OOP

✅ **Use abstract base classes (ABCs)** to enforce method implementation.
✅ **Follow the SOLID principles** (Single Responsibility, Open-Closed, Liskov, Interface Segregation, Dependency Inversion).
✅ **Leverage design patterns** for reusable and scalable solutions.
✅ **Use metaclasses sparingly** – they are powerful but can make code harder to understand.
✅ **Prefer composition over inheritance** when applicable.

Chapter 13: Real-World Object-Oriented Programming (OOP) Projects in Python

In this chapter, we will apply everything you've learned so far by building **real-world Python projects** using Object-Oriented Programming (OOP) principles. These projects will help you understand how to structure large codebases, manage complexity, and build scalable applications.

13.1 Building a Simple E-Commerce System

Overview:

An e-commerce system allows users to view products, add them to the cart, and proceed to checkout. We'll create a simplified version of this system using OOP principles.

Class Breakdown:

1. **Product** - Represents a product in the system.
2. **Cart** - Represents a shopping cart where products can be added.
3. **Order** - Represents an order, which includes products and customer details.

Code Example:

Product Class

```
python
class Product:
    def __init__(self, name, price, stock):
        self.name = name
        self.price = price
        self.stock = stock

    def __str__(self):
        return f"{self.name} - ${self.price} (Stock: {self.stock})"
```

Cart Class

```
python
class Cart:
    def __init__(self):
        self.items = []

    def add_product(self, product, quantity):
        if product.stock >= quantity:
            self.items.append({'product': product, 'quantity': quantity})
            product.stock -= quantity
            print(f"{quantity} x {product.name} added to cart.")
        else:
            print(f"Insufficient stock for {product.name}. Available:
{product.stock}")

    def total_price(self):
        return sum(item['product'].price * item['quantity'] for item in
self.items)

    def display_cart(self):
        for item in self.items:
            print(f"{item['product'].name} - ${item['product'].price} x
{item['quantity']}")
```

Order Class

```
python
class Order:
    def __init__(self, cart, customer_name, address):
        self.cart = cart
        self.customer_name = customer_name
        self.address = address
        self.order_id = self.generate_order_id()

    def generate_order_id(self):
        return f"ORD{1000 + len(self.cart.items)}"

    def complete_order(self):
        print(f"Order {self.order_id} for {self.customer_name} is being
processed.")
        print(f"Shipping to: {self.address}")
        print(f"Total Price: ${self.cart.total_price()}")
```

Usage Example:

```
python
# Create products
product1 = Product("Laptop", 1200, 10)
product2 = Product("Phone", 700, 5)

# Create cart and add products
cart = Cart()
cart.add_product(product1, 2)
cart.add_product(product2, 1)

# Display cart and total price
cart.display_cart()
print(f"Total Price: ${cart.total_price()}")

# Create and complete order
```

```
order = Order(cart, "John Doe", "123 Street, City")
order.complete_order()
```

Key Concepts Used:

- **Encapsulation:** Products, cart, and orders are represented as objects.
- **Inheritance:** You could extend `Product` to create specialized product types (e.g., `DigitalProduct`, `PhysicalProduct`).
- **Composition:** The `Order` class has a `Cart` as an attribute, showing a "has-a" relationship.

13.2 Building a Personal Task Manager

Overview:

A task manager helps users track their to-do lists. We'll build a simple task manager where users can create, view, update, and delete tasks.

Class Breakdown:

1. **Task** - Represents a single task.
2. **TaskManager** - Manages a list of tasks.

Code Example:

Task Class

```
python
class Task:
    def __init__(self, title, description, deadline, priority):
        self.title = title
        self.description = description
        self.deadline = deadline
        self.priority = priority
        self.completed = False

    def mark_complete(self):
        self.completed = True

    def __str__(self):
        status = "Completed" if self.completed else "Pending"
        return f"{self.title} ({status}) - Deadline: {self.deadline},
Priority: {self.priority}"
```

TaskManager Class

```
python
class TaskManager:
    def __init__(self):
        self.tasks = []

    def add_task(self, title, description, deadline, priority):
        task = Task(title, description, deadline, priority)
        self.tasks.append(task)
```

```
        print(f"Task '{title}' added.")

    def view_tasks(self):
        for task in self.tasks:
            print(task)

    def update_task(self, task_index, **kwargs):
        task = self.tasks[task_index]
        for key, value in kwargs.items():
            setattr(task, key, value)
        print(f"Task '{task.title}' updated.")

    def delete_task(self, task_index):
        task = self.tasks.pop(task_index)
        print(f"Task '{task.title}' deleted.")
```

Usage Example:

```
python
# Create task manager
task_manager = TaskManager()

# Add tasks
task_manager.add_task("Finish Report", "Complete the quarterly report",
"2025-03-01", "High")
task_manager.add_task("Buy Groceries", "Buy groceries for the week", "2025-
02-20", "Medium")

# View tasks
task_manager.view_tasks()

# Update a task
task_manager.update_task(0, completed=True)

# View tasks again after update
task_manager.view_tasks()

# Delete a task
task_manager.delete_task(1)
task_manager.view_tasks()
```

Key Concepts Used:

- **Encapsulation:** Tasks have attributes like title, description, and status encapsulated within the `Task` class.
- **Methods:** The `TaskManager` class provides various methods to interact with tasks.

13.3 Building a Simple Banking System

Overview:

A banking system where users can create accounts, deposit and withdraw money, and view their balance.

Class Breakdown:

1. **BankAccount** - Represents a user's bank account.
2. **Bank** - Manages a collection of bank accounts.

Code Example:

BankAccount Class

```
python
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
            print(f"Deposited ${amount}. New balance: ${self.balance}")
        else:
            print("Deposit amount must be positive.")

    def withdraw(self, amount):
        if amount > 0 and amount <= self.balance:
            self.balance -= amount
            print(f"Withdrew ${amount}. New balance: ${self.balance}")
        else:
            print("Insufficient funds or invalid amount.")

    def check_balance(self):
        return f"Balance for {self.owner}: ${self.balance}"
```

Bank Class

```
python
class Bank:
    def __init__(self):
        self.accounts = {}

    def create_account(self, owner, initial_balance=0):
        if owner not in self.accounts:
            self.accounts[owner] = BankAccount(owner, initial_balance)
            print(f"Account created for {owner}.")
        else:
            print(f"Account already exists for {owner}.")

    def get_account(self, owner):
        return self.accounts.get(owner, None)
```

Usage Example:

```
python
# Create bank and accounts
bank = Bank()
bank.create_account("Alice", 1000)
bank.create_account("Bob", 500)

# Deposit, withdraw, and check balance
alice_account = bank.get_account("Alice")
alice_account.deposit(200)
```

```
alice_account.withdraw(150)
print(alice_account.check_balance())

bob_account = bank.get_account("Bob")
bob_account.deposit(300)
bob_account.withdraw(100)
print(bob_account.check_balance())
```

Key Concepts Used:

- **Encapsulation:** Bank account attributes such as balance and owner are encapsulated within the `BankAccount` class.
- **Composition:** The `Bank` class manages a collection of `BankAccount` objects, showing a "has-a" relationship.

13.4 Best Practices for Real-World Projects

☑ **Plan your classes and relationships carefully** – Always define clear responsibilities.
☑ **Use OOP principles to make the system flexible and extensible** – E.g., inheritance, composition.
☑ **Follow the SOLID principles** to ensure maintainability.
☑ **Use design patterns** where appropriate to solve common design problems efficiently.