Object Oriented Programming in Python

Great! Let's go deeper into the **types of Object-Oriented Programming (OOP) concepts** in Python. I'll explain each type with examples, and then we can add assessments for practice.

**1. Class and Object**

A **class** is a blueprint for creating objects, while an **object** is an instance of a class.

**Example:**

class Student:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def display(self):

print(f"Student Name: {self.name}, Age: {self.age}")

# Creating an object

s1 = Student("Alice", 20)

s1.display()

**Types of Classes:**

1. **Regular Class** - Standard Python classes.
2. **Abstract Class** - Classes that cannot be instantiated (explained later).
3. **Singleton Class** - Ensures only one instance exists.

**Practice Task:**

Create a Book class with attributes title and author. Add a method details() to display the book information.

**2. Encapsulation (Data Hiding)**

Encapsulation **restricts direct access** to data using private/protected attributes.

**Types of Encapsulation:**

1. **Public attributes** (self.name) → Accessible from anywhere.
2. **Protected attributes** (self.\_name) → Should be accessed only within the class or subclasses.
3. **Private attributes** (self.\_\_name) → Can only be accessed within the class.

**Example:**

class BankAccount:

def \_\_init\_\_(self, account\_number, balance):

self.account\_number = account\_number # Public

self.\_bank\_name = "ABC Bank" # Protected

self.\_\_balance = balance # Private

def get\_balance(self):

return self.\_\_balance # Accessing private attribute

def deposit(self, amount):

if amount > 0:

self.\_\_balance += amount

print(f"Deposited: {amount}, New Balance: {self.\_\_balance}")

# Creating an object

acc = BankAccount("12345", 1000)

print(acc.account\_number) # Allowed (Public)

print(acc.\_bank\_name) # Allowed (Protected)

# print(acc.\_\_balance) # Error! (Private)

# Accessing private attribute via a method

print("Balance:", acc.get\_balance())

**Practice Task:**

Modify the Book class to include a private attribute \_price and create getter/setter methods for it.

**3. Inheritance (Reusability)**

Inheritance allows a child class to inherit properties from a parent class.

**Types of Inheritance:**

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

**Example: Single Inheritance**

class Animal:

def \_\_init\_\_(self, name):

self.name = name

def speak(self):

print("Animal makes a sound")

# Inheriting from Animal

class Dog(Animal):

def speak(self):

print(f"{self.name} barks!")

# Creating an object

dog = Dog("Buddy")

dog.speak()

**Example: Multiple Inheritance**

class Parent1:

def show1(self):

print("This is Parent 1")

class Parent2:

def show2(self):

print("This is Parent 2")

class Child(Parent1, Parent2):

def show\_child(self):

print("This is Child")

obj = Child()

obj.show1()

obj.show2()

obj.show\_child()

**Practice Task:**

Modify the Book class to create a subclass EBook that adds an attribute file\_size and a method download().

**4. Polymorphism (Same Function, Different Behavior)**

Polymorphism allows methods to be used in **different ways** depending on the object.

**Types of Polymorphism:**

1. **Method Overriding** – A subclass modifies an inherited method.
2. **Method Overloading** – Multiple methods with the same name but different arguments (Python does not support true overloading).
3. **Operator Overloading** – Customizing how operators like +, -, \* work.

**Example: Method Overriding**

class Bird:

def fly(self):

print("Birds can fly")

class Penguin(Bird):

def fly(self):

print("Penguins cannot fly")

# Creating objects

sparrow = Bird()

penguin = Penguin()

sparrow.fly()

penguin.fly()

**Example: Operator Overloading**

class Number:

def \_\_init\_\_(self, value):

self.value = value

def \_\_add\_\_(self, other):

return self.value + other.value

n1 = Number(10)

n2 = Number(20)

print(n1 + n2) # Calls \_\_add\_\_()

**Practice Task:**

Create a Shape class with a calculate\_area() method. Implement two subclasses Circle and Rectangle, each with its own calculate\_area().

**5. Abstraction (Hiding Implementation)**

Abstraction is the process of **hiding complex implementation details** and showing only essential features.

**Types of Abstraction:**

1. **Abstract Classes** – Cannot be instantiated, only used as a base class.
2. **Interfaces** – Enforces method implementation in child classes.

**Example: Abstract Class**

from abc import ABC, abstractmethod

class Vehicle(ABC): # Abstract class

@abstractmethod

def max\_speed(self):

pass

class Car(Vehicle):

def max\_speed(self):

return "Max speed is 200 km/h"

class Bike(Vehicle):

def max\_speed(self):

return "Max speed is 150 km/h"

car = Car()

bike = Bike()

print(car.max\_speed())

print(bike.max\_speed())

**Practice Task:**

Create an abstract class Payment with an abstract method process\_payment(). Implement two subclasses: CreditCardPayment and UPIPayment that define process\_payment().

**Conclusion**

|  |  |  |
| --- | --- | --- |
| **OOP Concept** | **Definition** | **Example** |
| **Class & Object** | Blueprint & Instance | Car, Student |
| **Encapsulation** | Hiding data | Private & Protected attributes |
| **Inheritance** | Reusing code | Parent-Child classes |
| **Polymorphism** | Same method, different behavior | Method Overriding, Overloading |
| **Abstraction** | Hiding implementation | Abstract classes |

**Final Challenge**

💡 **Design an OOP-based Python program for a "Library Management System" using all OOP concepts.**

* **Encapsulation:** Hide book prices.
* **Inheritance:** Book → EBook.
* **Polymorphism:** Override show\_details() in EBook.
* **Abstraction:** Create Library as an abstract class.

Let's break down the **Library Management System** project using OOP concepts in detail. This will help you understand how each concept fits into a real-world application.

**📌 System Requirements**

We need to create a **Library Management System** where users can:

1. **Add books** to the library (physical and e-books).
2. **Borrow and return books**.
3. **View book details**.
4. **Restrict access** to private attributes like price.
5. **Use different types of books** (e.g., Paper Books and EBooks).
6. **Enforce method implementation** for managing books.

**📌 Applying OOP Concepts**

|  |  |
| --- | --- |
| **OOP Concept** | **Usage in Library System** |
| **Class & Object** | Book, EBook, Library, User |
| **Encapsulation** | Hide book price using private attributes |
| **Inheritance** | EBook class inherits from Book |
| **Polymorphism** | Different show\_details() implementations for Book and EBook |
| **Abstraction** | Abstract class Library with abstract methods |

**📌 Implementation of OOP Concepts**

**1️⃣ Abstract Class (Abstraction)**

We'll create a **Library** class that defines essential operations for a library but does not provide implementations.

from abc import ABC, abstractmethod

class Library(ABC):

@abstractmethod

def add\_book(self, book):

pass

@abstractmethod

def borrow\_book(self, title):

pass

@abstractmethod

def return\_book(self, title):

pass

**Why use an Abstract Class?**

* Ensures all library types (e.g., PublicLibrary, UniversityLibrary) implement add\_book, borrow\_book, and return\_book.

**2️⃣ Encapsulation (Data Hiding)**

We'll **hide book prices** and provide getter and setter methods for controlled access.

class Book:

def \_\_init\_\_(self, title, author, price):

self.title = title

self.author = author

self.\_\_price = price # Private Attribute

def get\_price(self):

return self.\_\_price

def set\_price(self, new\_price):

if new\_price > 0:

self.\_\_price = new\_price

else:

print("Price must be positive!")

def show\_details(self):

print(f"Book: {self.title}, Author: {self.author}, Price: ${self.\_\_price}")

**Why use private attributes?**

* Prevents direct modification of the price.

**3️⃣ Inheritance (Code Reusability)**

The EBook class inherits from Book and adds a new feature: file\_size.

class EBook(Book):

def \_\_init\_\_(self, title, author, price, file\_size):

super().\_\_init\_\_(title, author, price)

self.file\_size = file\_size # Additional attribute for EBooks

def show\_details(self): # Method Overriding

print(f"EBook: {self.title}, Author: {self.author}, File Size: {self.file\_size}MB, Price: ${self.get\_price()}")

**Why use inheritance?**

* Reuses Book attributes while adding new ones (file\_size).

**4️⃣ Polymorphism (Different Behaviors)**

We override the show\_details() method to **display different details** for Book and EBook.

book1 = Book("Python Basics", "John Doe", 30)

ebook1 = EBook("Machine Learning", "Jane Smith", 25, 5)

book1.show\_details()

ebook1.show\_details()

**Output:**

Book: Python Basics, Author: John Doe, Price: $30

EBook: Machine Learning, Author: Jane Smith, File Size: 5MB, Price: $25

**5️⃣ Implementing Library System**

We'll create a PublicLibrary class implementing the Library abstract methods.

class PublicLibrary(Library):

def \_\_init\_\_(self):

self.books = [] # List to store books

def add\_book(self, book):

self.books.append(book)

print(f"Book '{book.title}' added to the library.")

def borrow\_book(self, title):

for book in self.books:

if book.title == title:

self.books.remove(book)

print(f"You have borrowed '{title}'.")

return book

print(f"Sorry, '{title}' is not available.")

return None

def return\_book(self, book):

self.books.append(book)

print(f"Book '{book.title}' has been returned.")

def display\_books(self):

print("\nAvailable Books in Library:")

for book in self.books:

book.show\_details()

**6️⃣ Using the Library System**

Now, let's **test our system**.

# Creating a library

library = PublicLibrary()

# Adding books

book1 = Book("Python Basics", "John Doe", 30)

ebook1 = EBook("Machine Learning", "Jane Smith", 25, 5)

library.add\_book(book1)

library.add\_book(ebook1)

# Display books

library.display\_books()

# Borrowing a book

borrowed\_book = library.borrow\_book("Python Basics")

# Returning the book

if borrowed\_book:

library.return\_book(borrowed\_book)

# Display books again

library.display\_books()

**📌 Expected Output**

Book 'Python Basics' added to the library.

Book 'Machine Learning' added to the library.

Available Books in Library:

Book: Python Basics, Author: John Doe, Price: $30

EBook: Machine Learning, Author: Jane Smith, File Size: 5MB, Price: $25

You have borrowed 'Python Basics'.

Book 'Python Basics' has been returned.

Available Books in Library:

Book: Python Basics, Author: John Doe, Price: $30

EBook: Machine Learning, Author: Jane Smith, File Size: 5MB, Price: $25

**📌 Additional Challenges**

1. **User Management:**
   * Add a User class with attributes like name and borrowed\_books.
   * Track which user borrows which book.
2. **Fine System:**
   * Add a system to calculate fines if a book is not returned on time.
3. **Multiple Library Types:**
   * Implement UniversityLibrary and DigitalLibrary with different borrowing rules.
4. **More Book Categories:**
   * Add more subclasses like ReferenceBook or Magazine.

**🎯 Key Takeaways**

✅ **Encapsulation:** Protected book price using private attributes.  
✅ **Inheritance:** EBook class inherited from Book.  
✅ **Polymorphism:** show\_details() was overridden.  
✅ **Abstraction:** Library defined an abstract blueprint.

This project follows **real-world principles** of OOP! 🚀

**📌 Practical Questions and Answer**

Certainly! Here are **some big tech interview questions** that will assess your understanding of Object-Oriented Programming (OOP) concepts:

**1. Design a Library Management System Using OOP Principles**

**Question:** Design a simple Library Management System that allows users to:

* Add books to the library
* Check out books
* Return books
* Track the status of books (whether available or checked out)
* Display information about the books in the library (e.g., title, author, publication year)

**Requirements:**

* Use **classes** to represent books, users, and the library.
* Use **inheritance** to model different types of users (e.g., regular users and librarians).
* Implement **polymorphism** to allow different types of books (e.g., physical books and e-books) to have different behaviours.
* Apply **encapsulation** to protect the data (e.g., the number of books checked out should not be accessible directly).
* Use **abstraction** for common library methods (e.g., check\_out() and return\_book()).

**Assessing OOP Concepts:**

* **Encapsulation**: Data is hidden within the objects, and only specific methods (getters/setters) interact with it.
* **Inheritance**: Different types of users and books can inherit from base classes.
* **Polymorphism**: Different books (physical, e-book) can behave differently in the system.
* **Abstraction**: Simplifying complex interactions like checking out a book into an abstract method for users and books.

**Tips for Answering:**

* **Start with the class diagram**: Before writing code, sketch out the relationships between the classes.
* **Break down the problem**: Identify the core components of the system, such as Book, User, Transaction, etc., and consider their interactions.
* **Explain OOP concepts**: As you build your solution, clearly explain how you're applying each of the OOP principles to your design (e.g., "I'm using inheritance here to allow different types of users to share common behavior...").

These questions are great for showcasing your OOP knowledge in action, as they require a blend of **design thinking** and **coding skills** to implement well-organized, maintainable systems.

Certainly! Let's break down the **Library Management System** problem using Object-Oriented Programming (OOP) concepts. I'll walk you through the steps and approach for solving it.

**Problem Recap:**

Design a Library Management System that allows users to:

* Add books to the library
* Check out books
* Return books
* Track the status of books (available or checked out)
* Display information about the books in the library

**Step-by-Step Approach:**

**1. Identify the Classes (Entities) in the System:**

The first step is to identify the **entities** or **objects** involved in the system.

* **Book**: Represents a book in the library.
* **User**: Represents a library user (who can check out books).
* **Library**: Represents the entire library system, including its collection of books.
* **Librarian**: A type of user with additional responsibilities (like adding/removing books).

**2. Identify Key Operations for Each Class:**

Each class will have specific methods or functions related to its responsibilities. Here’s a breakdown:

* **Book**:
  + Attributes: title, author, publication\_year, status (available/checked out).
  + Methods:
    - get\_details(): Display book details.
    - check\_out(): Change status to checked out.
    - return\_book(): Change status to available.
* **User**:
  + Attributes: name, user\_id, checked\_out\_books (a list of books).
  + Methods:
    - check\_out\_book(book): Add book to checked\_out\_books and mark it as checked out.
    - return\_book(book): Remove book from checked\_out\_books and mark it as returned.
* **Librarian** (inherits from User):
  + Additional responsibilities: Add/Remove books.
  + Methods:
    - add\_book(book): Add a book to the library.
    - remove\_book(book): Remove a book from the library.
* **Library**:
  + Attributes: books (a list of books), users (a list of users).
  + Methods:
    - add\_user(user): Add a user to the system.
    - remove\_user(user): Remove a user from the system.
    - display\_books(): Display all books in the library.

**3. Design Relationships Between Classes (Inheritance):**

The **Librarian** class can **inherit** from the **User** class since a librarian is essentially a user with extra responsibilities.

**4. Define Core OOP Principles:**

Let’s apply OOP concepts to the design:

* **Encapsulation**:
  + Each class will encapsulate its data (e.g., Book class will have attributes like title, status and will manage them using methods).
  + Sensitive data, such as transaction history or book status, will be accessed only through specific methods.
* **Abstraction**:
  + We’ll hide the internal workings of methods like check\_out\_book and return\_book inside the respective classes.
  + Users don’t need to know how the book’s status is updated; they just interact with the methods.
* **Inheritance**:
  + **Librarian** inherits from **User** to reuse common functionality (like check\_out\_book and return\_book).
* **Polymorphism**:
  + A book can be checked out or returned, but the implementation of how a book behaves when checked out could vary if you have different types of books (like physical or e-books).

**5. Code Implementation:**

Let’s translate this approach into code:

# Step 1: Define the Book Class

class Book:

def \_\_init\_\_(self, title, author, publication\_year):

self.title = title

self.author = author

self.publication\_year = publication\_year

self.status = "Available" # The default status

def get\_details(self):

return f"{self.title} by {self.author} ({self.publication\_year}) - {self.status}"

def check\_out(self):

if self.status == "Available":

self.status = "Checked Out"

else:

print(f"The book '{self.title}' is already checked out.")

def return\_book(self):

if self.status == "Checked Out":

self.status = "Available"

else:

print(f"The book '{self.title}' wasn't checked out.")

# Step 2: Define the User Class

class User:

def \_\_init\_\_(self, name, user\_id):

self.name = name

self.user\_id = user\_id

self.checked\_out\_books = []

def check\_out\_book(self, book):

if book.status == "Available":

self.checked\_out\_books.append(book)

book.check\_out()

print(f"{self.name} checked out '{book.title}'")

else:

print(f"Cannot check out '{book.title}', it is already checked out.")

def return\_book(self, book):

if book in self.checked\_out\_books:

self.checked\_out\_books.remove(book)

book.return\_book()

print(f"{self.name} returned '{book.title}'")

else:

print(f"{self.name} hasn't checked out '{book.title}'.")

# Step 3: Define the Librarian Class (Inherits from User)

class Librarian(User):

def \_\_init\_\_(self, name, user\_id):

super().\_\_init\_\_(name, user\_id)

def add\_book(self, library, book):

library.add\_book(book)

print(f"Librarian {self.name} added '{book.title}' to the library.")

def remove\_book(self, library, book):

library.remove\_book(book)

print(f"Librarian {self.name} removed '{book.title}' from the library.")

# Step 4: Define the Library Class

class Library:

def \_\_init\_\_(self):

self.books = []

self.users = []

def add\_user(self, user):

self.users.append(user)

def remove\_user(self, user):

self.users.remove(user)

def add\_book(self, book):

self.books.append(book)

def remove\_book(self, book):

self.books.remove(book)

def display\_books(self):

for book in self.books:

print(book.get\_details())

# Step 5: Testing the System

# Create Library and Users

library = Library()

librarian = Librarian("Alice", 101)

user1 = User("John", 102)

# Add books to the library

book1 = Book("The Catcher in the Rye", "J.D. Salinger", 1951)

book2 = Book("1984", "George Orwell", 1949)

librarian.add\_book(library, book1)

librarian.add\_book(library, book2)

# Display all books

library.display\_books()

# User checks out a book

user1.check\_out\_book(book1)

# Display books after checkout

library.display\_books()

# User returns the book

user1.return\_book(book1)

# Display books after return

library.display\_books()

**6. Explanation of the Code:**

1. **Book Class**:
   * Manages the details of the book and its status (whether it's available or checked out). The methods check\_out() and return\_book() change the status of the book.
2. **User Class**:
   * A basic user who can check out and return books. The checked\_out\_books list tracks books currently checked out by the user.
3. **Librarian Class**:
   * Inherits from the User class but adds additional functionality to add or remove books from the library.
4. **Library Class**:
   * Maintains a collection of books and users. It can add users, remove users, and display books.

**7. Final Thoughts:**

By designing this system using OOP principles:

* **Encapsulation**: We’ve encapsulated book status within the Book class and only allow interaction through defined methods.
* **Inheritance**: The Librarian class inherits from User, reusing the functionality while extending it with new responsibilities.
* **Polymorphism**: If we had different types of books (e.g., e-books), we could override the check\_out() method to handle them differently.
* **Abstraction**: We hide the internal details of checking out or returning a book; the user just interacts with the methods check\_out\_book() and return\_book().

In the Similar way, solve the below problems,

**1. Design a Simple Payment Gateway System**

**Question:** Design a simple payment gateway system where users can:

* Add credit/debit card information
* Make payments
* Store transaction history

The system should:

* Support multiple payment methods (e.g., Credit card, PayPal, Bank transfer).
* Provide a payment status (Success/Failure).
* Keep track of transaction details (e.g., amount, transaction ID, payment method).
* Allow users to update their payment information.

**Requirements:**

* Use **abstract classes** for common payment methods.
* Use **inheritance** to define different types of payment methods (Credit Card, PayPal, Bank Transfer).
* Implement **polymorphism** for the payment processing logic (each payment method processes payments differently).
* Use **encapsulation** to ensure that sensitive data like card numbers or transaction IDs are protected.

**Assessing OOP Concepts:**

* **Abstraction**: Abstract class for payment methods hides the common processing logic.
* **Inheritance**: Each payment method (credit card, PayPal, etc.) will inherit from a common payment method class.
* **Polymorphism**: Different payment methods can have different implementations for the process\_payment() method.
* **Encapsulation**: Protect sensitive data like card information and transaction details using private fields and methods.

Sure! Here are three more questions that will assess your understanding of OOP concepts:

**2. Design a Bank Account System**

**Question:** Design a simple bank account system where users can:

* Create a bank account
* Deposit money
* Withdraw money
* Transfer money between accounts
* Check the balance
* Apply different interest rates based on account types (e.g., Savings Account, Checking Account).

**Requirements:**

* Use **inheritance** to define different types of accounts (Savings, Checking).
* Implement **polymorphism** to handle deposit and withdrawal methods differently based on account type.
* Use **encapsulation** to hide the balance from direct access.
* Use **abstraction** to define a common method for transaction processing.

**3. Design an Online Shopping Cart System**

**Question:** Design an online shopping cart system that allows:

* Adding products to the cart
* Removing products from the cart
* Updating product quantities
* Viewing the total cost of the items in the cart
* Checking out and applying discounts based on product categories

**Requirements:**

* Use **classes** for Products, Cart, and Users.
* Use **inheritance** to define different types of products (e.g., Electronics, Clothing).
* Implement **polymorphism** to calculate the discount for each product type.
* **Encapsulate** sensitive data like prices and quantities.
* **Abstract** the cart calculation logic so that it’s reusable for different scenarios.

**4. Design a Traffic Management System**

**Question:** Design a traffic management system where:

* Vehicles can enter and exit a parking lot.
* Vehicles can have different types (e.g., Sedan, Truck, Bike) with different parking fees.
* The system tracks the total income from the parking lot.
* The system can generate a report of vehicles currently in the lot.

**Requirements:**

* Use **inheritance** for different types of vehicles.
* Implement **polymorphism** to apply different parking fee calculations based on the vehicle type.
* Use **encapsulation** to protect the parking lot’s income and vehicle data.
* **Abstract** the parking fee calculation logic.

Let me know how you do with these, and feel free to ask for clarification or answers!