# 🐍 Object-Oriented Programming (OOP) in Python - Interview Preparation

---

### 🎯 Objective:

Prepare for Python interviews with an interactive approach to **Object-Oriented Programming (OOP)** by covering key concepts, hands-on examples, and frequently asked interview questions.

---

## ✅ 1. What is Object-Oriented Programming (OOP)?

**Object-oriented programming (OOP)** is a programming paradigm that uses objects and classes to organize code into reusable and manageable components.

---

## 📚 2. Key Concepts of OOP

---

### ➤ 1. Class and Object

- **Class:** A blueprint or template to create objects.
- **Object:** An instance of a class.

**📝 Example:**

```
# Define a class
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def display(self):
        print(f"This is a {self.brand} {self.model}")
```

```
# Create an object
car1 = Car("Toyota", "Corolla")
car1.display()
```

👉 *Output:*

```
This is a Toyota Corolla
```

---

## ➢ 2. Encapsulation

- **Encapsulation** is wrapping data and methods into a single unit (class).
- It restricts direct access to variables and methods to protect data.

📝 **Example:**

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.__age = age  # Private attribute

    def display(self):
        print(f"Name: {self.name}, Age: {self.__age}")

# Create an object
person1 = Person("Sanket", 30)
person1.display()

# Accessing private attribute directly will raise an error
# print(person1.__age)  # AttributeError
```

---

## ➢ 3. Inheritance

- **Inheritance** allows one class to inherit the properties and behavior of another class.

📝 **Types of Inheritance:**

1. **Single Inheritance**
2. **Multiple Inheritance**
3. **Multilevel Inheritance**
4. **Hierarchical Inheritance**
5. **Hybrid Inheritance**

💡 **Single Inheritance Example:**

```
class Parent:
    def show(self):
        print("Parent class")

class Child(Parent):
    def display(self):
        print("Child class")

obj = Child()
obj.show()
obj.display()
```

👉 *Output:*

```
Parent class
Child class
```

---

## ➤ 4. Polymorphism

- **Polymorphism** means having multiple forms. It allows methods in different classes to have the same name but behave differently.

💡 **Method Overriding Example:**

```
class Animal:
    def sound(self):
        print("Animals make different sounds")

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

class Cat(Animal):
    def sound(self):
        print("Cat meows")

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

dog.sound()  # Dog barks
cat.sound()  # Cat meows
```

---

## ➢ 5. Abstraction

- **Abstraction** hides implementation details and only shows essential features.

💡 **Abstract Class Example:**

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width
    def area(self):
        return self.length * self.width

rect = Rectangle(5, 3)
print("Area:", rect.area())
```

👉 *Output:*

```
Area: 15
```

---

## 🧠 3. Common Interview Questions on OOP

---

### 🔥 1. What is the difference between a class and an object?

- **Class:** A blueprint/template to create objects.
- **Object:** An instance of a class.

---

### 🔥 2. Can you explain multiple inheritance in Python?

- Multiple inheritance allows a class to inherit from more than one base class.

📝 **Example:**

```
class A:
    def method_A(self):
        print("Method from Class A")

class B:
    def method_B(self):
        print("Method from Class B")

class C(A, B):
    def method_C(self):
        print("Method from Class C")

obj = C()
obj.method_A()
obj.method_B()
obj.method_C()
```

👉 *Output:*

```
Method from Class A
Method from Class B
Method from Class C
```

---

## 🔥 3. What is the difference between method overloading and method overriding?

- **Method Overloading:** Same method name but a different number of parameters. (Not natively supported in Python.)
- **Method Overriding:** A child class redefines a parent class method with the same signature.

---

## 🔥 4. Can you create a private method in Python?

- Yes, by using a double underscore `__` before the method name.

📝 **Example:**
t

```
class MyClass:
    def __private_method(self):
        print("This is a private method")

    def public_method(self):
        self.__private_method()
```

```
obj = MyClass()
obj.public_method()

# obj.__private_method()  # Raises AttributeError
```

---

### 🔥 5. What is `super()` in Python?

- `super()` is used to call the parent class constructor or method.

📝 **Example:**

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

class Child(Parent):
    def __init__(self, name, age):
        super().__init__(name)
        self.age = age

obj = Child("Sanket", 30)
print(obj.name, obj.age)
```

👉 *Output:*

```
Sanket 30
```

---

## 🕹️ 4. Practice Challenges

---

### 🚀 1. Create a Class for Employee and Manager

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

    def display(self):
        print(f"Employee: {self.name}, Salary: {self.salary}")

class Manager(Employee):
    def __init__(self, name, salary, department):
        super().__init__(name, salary)
        self.department = department

    def display(self):
        print(f"Manager: {self.name}, Salary: {self.salary},
Department: {self.department}")

emp = Employee("John", 50000)
mgr = Manager("Alice", 80000, "HR")

emp.display()
mgr.display()
```

---

### 🚀 2. Implement Multiple Inheritance with Example

```
class A:
    def method_A(self):
        print("Method from Class A")

class B:
    def method_B(self):
        print("Method from Class B")

class C(A, B):
    def method_C(self):
        print("Method from Class C")

obj = C()
obj.method_A()
obj.method_B()
obj.method_C()
```

---

### 🚀 3. Create a Banking System with Deposit and Withdrawal

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

    def deposit(self, amount):
        self.balance += amount
        print(f"Deposited {amount}. New Balance: {self.balance}")

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

# Create account
account = BankAccount("Sanket", 1000)
account.deposit(500)
account.withdraw(300)
account.withdraw(1500)
```

---

## 🎁 5. Advanced Concepts in OOP

---

### ➢ 1. Operator Overloading

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

p1 = Point(1, 2)
p2 = Point(3, 4)
result = p1 + p2
print(f"Result: ({result.x}, {result.y})")
```

👉 *Output:*

```
Result: (4, 6)
```

---

### ➤ 2. Class Method and Static Method

```
class MyClass:
    class_variable = "Class Level Variable"

    @classmethod
    def class_method(cls):
        print("Class Method:", cls.class_variable)

    @staticmethod
    def static_method():
        print("Static Method: No Access to Class Variables")

MyClass.class_method()
MyClass.static_method()
```

---

### ➤ 3. Multiple Constructor using `@classmethod`

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    @classmethod
    def from_string(cls, student_string):
        name, age = student_string.split(",")
        return cls(name, int(age))

s1 = Student.from_string("John,25")
print(s1.name, s1.age)
```

---

## 🎯 6. Quick Tips for Interview Success

- ✅ Understand all four pillars of OOP (Encapsulation, Inheritance, Polymorphism, Abstraction).
- ✅ Be comfortable with class, object, and method creation.
- ✅ Know how to use `super()` and override parent class methods.
- ✅ Learn operator overloading and advanced class features.
- ✅ Practice with real-world examples.

---