Advanced Object-Oriented Programming (OOP) concepts in Python Inheritance, Polymorphism, and Encapsulation with real-world examples.

Day 15: Object-Oriented Programming - Advanced

Python's **Object-Oriented Programming (OOP)** provides a way to structure code using objects and classes. Today, we will focus on **advanced OOP concepts**:

- **Inheritance** (Reusing code from a parent class)
- **Polymorphism** (Using a common interface for multiple behaviors)
- **Encapsulation** (Restricting access to methods and variables for data security)

Let's dive into each topic in detail with practical examples.

---

1. Inheritance (Code Reusability & Hierarchical Structure)

Definition:
Inheritance allows a class (**child class**) to acquire properties and behaviors (methods) from another class (**parent class**). This promotes **code reusability** and creates a natural hierarchy in classes.

Types of Inheritance in Python

Python supports multiple types of inheritance:

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

1.1 Single Inheritance (One Parent → One Child)

A child class inherits from a single parent class.

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

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

# Child class inherits from Animal
class Dog(Animal):
    def speak(self):  # Overriding parent method
        return f"{self.name} barks!"

dog = Dog("Buddy")
print(dog.speak())  # Output: Buddy barks!
```

👉 **Key Takeaways:**

- **Child class (Dog)** inherits properties from Animal.
- **Method overriding** allows child class to modify the inherited method.

1.2 Multiple Inheritance (One Child → Multiple Parents)

A class can inherit from multiple parent classes.

```
class Engine:
    def start_engine(self):
        return "Engine started"

class Wheels:
    def rotate_wheels(self):
        return "Wheels are rotating"

# Car inherits from both Engine and Wheels
class Car(Engine, Wheels):
    def drive(self):
        return "Car is driving"

my_car = Car()
print(my_car.start_engine())  # Output: Engine started
print(my_car.rotate_wheels()) # Output: Wheels are rotating
print(my_car.drive())         # Output: Car is driving
```

👉 **Key Takeaways:**

- **Multiple inheritance** allows sharing behavior from multiple parent classes.
- Python resolves conflicts using the **Method Resolution Order (MRO)**.

1.3 Multilevel Inheritance (Grandparent → Parent → Child)

A class is derived from another derived class.

```
class Vehicle:
    def vehicle_info(self):
        return "This is a vehicle"


class Car(Vehicle):
    def car_info(self):
        return "This is a car"


class ElectricCar(Car):
    def battery_info(self):
        return "Battery capacity is 100 kWh"


tesla = ElectricCar()
print(tesla.vehicle_info())  # Output: This is a vehicle
print(tesla.car_info())      # Output: This is a car
print(tesla.battery_info())  # Output: Battery capacity is 100 kWh
```

👉 **Key Takeaways:**

- Inherits properties in a **hierarchical** fashion.
- Used when classes have **progressive specialization**.

1.4 Hierarchical Inheritance (One Parent → Multiple Children)

One parent class has multiple child classes.

```
class Animal:
    def make_sound(self):
        return "Some generic sound"

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

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

cat = Cat()
dog = Dog()
print(cat.make_sound())  # Output: Meow!
print(dog.make_sound())  # Output: Bark!
```

👉 **Key Takeaways:**

- Multiple child classes **inherit** from the same parent.
- Each child class **modifies** the inherited behavior.

---

2. Polymorphism (Same Interface, Different Behavior)

Definition:
Polymorphism allows different classes to be treated as instances of the same class through a **common interface**.

2.1 Method Overriding (Runtime Polymorphism)

A child class provides its **own implementation** of a method already defined in the parent class.

```
class Bird:
    def fly(self):
        return "Some birds can fly"

class Sparrow(Bird):
    def fly(self):
        return "Sparrow can fly"

class Penguin(Bird):
    def fly(self):
        return "Penguins cannot fly"

birds = [Sparrow(), Penguin()]

for bird in birds:
    print(bird.fly())

# Output:
# Sparrow can fly
```

Penguins cannot fly

👉 **Key Takeaways:**

- The **same method (fly)** behaves **differently** in different classes.

2.2 Method Overloading (Compile-time Polymorphism - Python Alternative)

Python does not support **true** method overloading, but we can achieve it using **default arguments**.

```
class MathOperations:
  def add(self, a, b, c=0):
    return a + b + c


math_op = MathOperations()
print(math_op.add(5, 10))    # Output: 15
print(math_op.add(5, 10, 20)) # Output: 35
```

👉 **Key Takeaways:**

- The method add() works with **different numbers of arguments**.

---

3. Encapsulation (Data Hiding & Restriction)

Definition:
Encapsulation hides **implementation details** and restricts direct access to variables using access specifiers:

- **Public (name)** → Accessible anywhere
- **Protected (_name)** → Accessible within class and subclasses
- **Private (__name)** → Accessible only within the class

3.1 Public Members

```
class Person:
    def __init__(self, name):
        self.name = name  # Public variable

p = Person("Alice")
print(p.name)  # Output: Alice
```

3.2 Protected Members

```
class Person:
    def __init__(self, name):
        self._name = name  # Protected variable

class Employee(Person):
    def show(self):
        return f"Employee Name: {self._name}"

emp = Employee("John")
print(emp.show())  # Output: Employee Name: John
```

3.3 Private Members

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

    def get_balance(self):
        return self.__balance

account = BankAccount(5000)
print(account.get_balance())  # Output: 5000
print(account.__balance)  # AttributeError: 'BankAccount' object has no attribute '__balance'
```

👉 **Key Takeaways:**

- **Private members (__balance)** cannot be accessed directly.

Conclusion

✅ **Inheritance** → Code reuse & hierarchy
✅ **Polymorphism** → Different behavior with the same method
✅ **Encapsulation** → Data hiding for security

These concepts make Python OOP **powerful & flexible**, leading to clean, maintainable, and scalable code!