Object Oriented Program

Object-Oriented Programming (OOP) in Python is a programming paradigm that uses objects and classes to organize code in a more modular and reusable way. OOP is built around the concepts of **classes**, **objects**, **inheritance**, **polymorphism**, **encapsulation**, and **abstraction**.

Here, I will provide examples that explain these concepts, including all types of polymorphism, inheritance, and more.

1. Class and Object

A **class** is a blueprint for creating objects (instances), and an **object** is an instance of a class.

Example: Basic Class and Object

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

    def bark(self):
        return f"{self.name} says Woof!"

# Creating an object of the Dog class
my_dog = Dog("Buddy", "Golden Retriever")

# Accessing object's attributes and methods
print(my_dog.name)  # Output: Buddy
print(my_dog.bark())  # Output: Buddy says Woof!
```

- __init__() is the constructor method that initializes object attributes.
- self refers to the instance of the class.

---

2. Inheritance

Inheritance allows a class (child class) to inherit methods and properties from another class (parent class). This promotes code reuse.

Example: Inheritance

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

    def sound(self):
        return "Some generic animal sound"

class Dog(Animal):  # Inherits from Animal class
    def sound(self):  # Method Overriding
        return f"{self.name} says Woof!"

class Cat(Animal):  # Inherits from Animal class
    def sound(self):  # Method Overriding
        return f"{self.name} says Meow!"

# Create objects of Dog and Cat
dog = Dog("Buddy")
cat = Cat("Whiskers")

print(dog.sound())  # Output: Buddy says Woof!
print(cat.sound())  # Output: Whiskers says Meow!
```

- The Dog and Cat classes inherit from the Animal class.
- Both Dog and Cat override the sound() method from the parent class.

3. Polymorphism

Polymorphism allows one method to behave differently depending on the object that calls it. There are two types of polymorphism: **Method Overriding** and **Method Overloading**.

3.1 Method Overriding (Run-time Polymorphism)

Method overriding occurs when a subclass provides a specific implementation of a method that is already defined in the parent class.

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

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

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

# Polymorphism: The same method behaves differently based on object type
def make_sound(animal):
    animal.sound()

dog = Dog()
cat = Cat()

make_sound(dog)  # Output: Dog barks
make_sound(cat)  # Output: Cat meows
```

- The sound() method is overridden in both Dog and Cat classes.

3.2 Method Overloading (Compile-time Polymorphism)

Python does not support traditional method overloading as in other languages (e.g., C++, Java). However, we can simulate method overloading by checking the number of arguments passed to a method.

```
class Math:
    def add(self, a, b=None):
        if b is None:
            return a + a  # if only one argument is provided, add it to itself
        else:
            return a + b  # if two arguments are provided, add them

math = Math()
print(math.add(5))     # Output: 10 (Overloading behavior)
print(math.add(5, 3))  # Output: 8
```

- In this example, the add() method behaves differently based on the number of arguments provided.

---

4. Encapsulation

Encapsulation is the concept of restricting access to certain details of an object and providing access only through public methods.

Example: Encapsulation

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

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
        else:
            print("Insufficient funds")

    def get_balance(self):
        return self.__balance

# Creating object of BankAccount
account = BankAccount("John", 1000)

# Accessing methods to deposit and withdraw
account.deposit(500)
account.withdraw(200)

# Accessing the private balance using the getter method
print(account.get_balance())  # Output: 1300

# Direct access to the private variable will raise an error
# print(account.__balance)  # AttributeError: 'BankAccount' object has no attribute '__balance'
```

- The __balance attribute is private and cannot be accessed directly from outside the class. Methods like deposit() and withdraw() provide controlled access.

---

5. Abstraction

Abstraction involves hiding the complex implementation details and showing only the essential features of an object.

Example: Abstraction Using ABC (Abstract Base Class)

```
from abc import ABC, abstractmethod

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

class Dog(Animal):
    def sound(self):
        return "Woof!"

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

# Cannot instantiate abstract class Animal
# animal = Animal()  # This will raise an error

dog = Dog()
cat = Cat()

print(dog.sound())  # Output: Woof!
print(cat.sound())  # Output: Meow!
```

- The Animal class is abstract, and you cannot create an instance of it directly. The sound() method is abstract, meaning it must be implemented by any subclass (like Dog or Cat).