**Object-Oriented Programming (OOP) Concepts in Python: Abstraction, Encapsulation, Inheritance, and Polymorphism**

Python is an object-oriented programming language, meaning it provides structures to implement OOP principles like **Abstraction**, **Encapsulation**, **Inheritance**, and **Polymorphism**. These concepts help improve the structure, organization, and reusability of code.

Let's break down each concept with detailed explanations and examples.

**1. Abstraction**

**Abstraction** is the concept of hiding the implementation details and showing only the essential features of an object. In Python, abstraction is typically achieved using abstract classes and methods (using the abc module).

**Example of Abstraction:**

from abc import ABC, abstractmethod

# Abstract Class

class Vehicle(ABC):

@abstractmethod

def start\_engine(self):

pass

@abstractmethod

def stop\_engine(self):

pass

# Concrete Class (inheriting the abstract class)

class Car(Vehicle):

def start\_engine(self):

return "Car engine started"

def stop\_engine(self):

return "Car engine stopped"

class Bike(Vehicle):

def start\_engine(self):

return "Bike engine started"

def stop\_engine(self):

return "Bike engine stopped"

# Instantiate objects of Car and Bike

car = Car()

bike = Bike()

print(car.start\_engine()) # Output: Car engine started

print(bike.start\_engine()) # Output: Bike engine started

**Explanation:**

* **Vehicle** is an abstract class that defines two abstract methods (start\_engine and stop\_engine), meaning the methods are declared but not implemented.
* **Car** and **Bike** are concrete classes that inherit from Vehicle and must implement the abstract methods.
* The user interacts with high-level concepts (start or stop the engine) without worrying about the specific implementations.

**2. Encapsulation**

**Encapsulation** refers to the bundling of data (attributes) and methods (functions) that operate on that data within one unit or class. It also involves restricting direct access to some of an object's components, which is done through access modifiers such as public, protected, and private.

**Example of Encapsulation:**

class BankAccount:

def \_\_init\_\_(self, owner, balance):

self.owner = owner # Public attribute

self.\_\_balance = balance # Private attribute (encapsulated)

def deposit(self, amount):

self.\_\_balance += amount

def withdraw(self, amount):

if self.\_\_balance >= amount:

self.\_\_balance -= amount

else:

print("Insufficient balance")

def get\_balance(self):

return self.\_\_balance

# Creating an object of BankAccount

account = BankAccount("Alice", 1000)

# Accessing public attribute

print(account.owner) # Output: Alice

# Accessing private attribute will raise an error

# print(account.\_\_balance) # AttributeError: 'BankAccount' object has no attribute '\_\_balance'

# Correct way to access private attribute (through a method)

print(account.get\_balance()) # Output: 1000

# Deposit money

account.deposit(500)

print(account.get\_balance()) # Output: 1500

**Explanation:**

* \_\_balance is a **private attribute** and cannot be accessed directly from outside the class. It is protected from unauthorized access.
* Methods deposit, withdraw, and get\_balance provide controlled access to this private attribute.
* Encapsulation hides the internal state of the object and only allows access through methods, ensuring that sensitive data is properly protected.

**3. Inheritance**

**Inheritance** is a mechanism where one class (called the child class) inherits attributes and methods from another class (called the parent class). This promotes code reusability and logical hierarchy.

**Example of Inheritance:**

python

Copy code

# Parent class

class Animal:

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

self.name = name

def speak(self):

return f"{self.name} makes a sound."

# Child class inheriting from Animal

class Dog(Animal):

def speak(self):

return f"{self.name} barks."

# Another child class

class Cat(Animal):

def speak(self):

return f"{self.name} meows."

# Instantiate objects

dog = Dog("Buddy")

cat = Cat("Whiskers")

print(dog.speak()) # Output: Buddy barks.

print(cat.speak()) # Output: Whiskers meows.

**Explanation:**

* Animal is the **parent class** that has a method speak() which is inherited by both Dog and Cat.
* Both Dog and Cat **inherit** from Animal, but they override the speak() method to provide their specific implementation (barking and meowing, respectively).
* This allows the child classes to inherit common behavior (like initialization with a name) and customize the behavior when needed (overriding speak()).

**4. Polymorphism**

**Polymorphism** allows objects of different classes to be treated as objects of a common super class. It means "many forms", and in Python, it allows different classes to have methods with the same name but potentially different behavior.

**Example of Polymorphism:**

class Bird:

def sound(self):

return "Bird makes a sound."

class Sparrow(Bird):

def sound(self):

return "Sparrow chirps."

class Parrot(Bird):

def sound(self):

return "Parrot talks."

# Function demonstrating polymorphism

def make\_sound(bird):

print(bird.sound())

# Different objects can be passed to the same function

sparrow = Sparrow()

parrot = Parrot()

make\_sound(sparrow) # Output: Sparrow chirps.

make\_sound(parrot) # Output: Parrot talks.

**Explanation:**

* Bird is the parent class, and both Sparrow and Parrot are child classes that override the sound() method.
* **Polymorphism** is achieved when different objects (Sparrow, Parrot) can be passed to the same function (make\_sound), and each object behaves according to its own method definition of sound().
* The function doesn’t need to know the specific type of object it’s handling; it simply calls the sound() method and relies on the object to execute the correct implementation.

**Conclusion**

1. **Abstraction**: Hides the complex implementation details from the user and only exposes the essential features. Achieved using abstract classes and methods.
2. **Encapsulation**: Restricts access to certain components of an object, preventing external access to private data and only allowing controlled interaction.
3. **Inheritance**: Allows a new class (child) to inherit properties and behaviors from an existing class (parent), promoting code reusability.
4. **Polymorphism**: Enables a single function or method to work with objects of different classes, allowing for flexible and dynamic code.

These OOP principles help in designing clean, organized, and reusable code in Python, making it easier to develop large-scale applications.