Object-Oriented Concepts in Python

October 07, 2025

Classes and Objects

Objective: To understand the fundamental concepts of classes and objects in Python by creating a simple Car class and instantiating its objects.

Theory: A **class** is a blueprint or a template for creating objects. It defines a set of attributes (data) and methods (functions) that the objects created from it will have. An **object** is a specific instance of a class. When a class is defined, no memory is allocated until an object is created.

Program:

```
# A class is a blueprint for objects.
class Car:
    # The __init__ method is a constructor, called when a new object is created.
    # It initializes the object's attributes.
    def __init__(self, make, model, year):
        self.make = make   # Attribute 1
        self.model = model # Attribute 2
        self.year = year   # Attribute 3
        print(f"A new {self.make} {self.model} has been created.")

    # A method is a function defined within a class.
    def display_info(self):
        print(f"Make: {self.make}")
        print(f"Model: {self.model}")
        print(f"Year: {self.year}")

# An object is an instance of a class.
# Here, we create two objects (instances) of the Car class.
car1 = Car("Toyota", "Camry", 2022)
car2 = Car("Honda", "Civic", 2023)
```

```
# We can access the object's methods and attributes using dot notation.
print("\n--- Car 1 Information ---")
car1.display_info()

print("\n--- Car 2 Information ---")
car2.display_info()
```

Output:

```
A new Toyota Camry has been created.
A new Honda Civic has been created.

--- Car 1 Information ---
Make: Toyota
Model: Camry
Year: 2022

--- Car 2 Information ---
Make: Honda
Model: Civic
Year: 2023
```

Inheritance

Objective: To demonstrate the principle of inheritance by creating a Dog class that inherits from a parent Animal class, thus reusing its methods and attributes.

Theory: Inheritance is a mechanism where a new class (child class) inherits the attributes and methods of an existing class (parent class). This promotes code reuse and creates a logical hierarchy. The super() function is often used to call the parent class's constructor (__init__) and other methods.

Types of Inheritance:

1. **Single Inheritance**: A derived class inherits from only one base class.
2. **Multiple Inheritance**: A derived class inherits from multiple base classes.

3. **Multilevel Inheritance**: A derived class inherits from a class that itself is a derived class, forming a chain (e.g., Class A -> Class B -> Class C).
4. **Hierarchical Inheritance**: Multiple derived classes inherit from a single base class.
5. **Hybrid Inheritance**: A combination of two or more types of inheritance, such as combining multilevel and multiple inheritance.

Polymorphism

Objective: To understand polymorphism by creating different classes that share the same method name but have different implementations.

Theory: Polymorphism (meaning "many forms") allows objects of different classes to be treated as objects of a common base class. This is often achieved through method overriding, where a child class provides its own unique implementation for a method already defined in its parent class.

Types of Polymorphism

Polymorphism in Python refers to ability of the same method or operation to behave differently based on object or context. It mainly includes **compile-time** and **runtime polymorphism**.

Encapsulation

Objective: To implement encapsulation by using "private" attributes to restrict direct access to an object's data.

Theory: Encapsulation is the principle of bundling data (attributes) and the methods that operate on that data into a single unit (the class). It also involves data hiding, where the internal state of an object is protected from direct external access. In Python, this is a convention using leading underscores. A single underscore (_) indicates a protected member, and a double underscore (__) "mangles" the name to make it harder to access from outside.

Program:

```
class BankAccount:
    def __init__(self, initial_balance):
        # A private attribute, indicated by the double underscore.
        self.__balance = initial_balance

    # A public method to deposit money.
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
```

```
            print(f"Deposited {amount}. New balance: {self.__balance}")
        else:
            print("Deposit amount must be positive.")

    # A public method to get the balance. This is the controlled way to access the data.
    def get_balance(self):
        return self.__balance

# Create a bank account object.
my_account = BankAccount(1000)

# We can access public methods to interact with the object.
my_account.deposit(500)
print(f"Current balance from get_balance(): {my_account.get_balance()}")

# This direct access attempt will fail (or raise an error) because `__balance` is "private".
# The interpreter "mangles" the name to `_BankAccount__balance`.
print("Trying to access private attribute directly...")
try:
    print(my_account.__balance)
except AttributeError as e:
    print(f"Error: {e}")
```

Output:

```
Deposited 500. New balance: 1500
Current balance from get_balance(): 1500
Trying to access private attribute directly...
Error: 'BankAccount' object has no attribute '__balance'
```

Abstraction

Objective: To understand abstraction by creating an abstract base class that defines a common interface for its derived classes.

Theory: Abstraction is the process of hiding complex implementation details and showing only the essential features of an object. In Python, this is achieved using **abstract base classes (ABC)** from the abc module. An abstract class defines methods that must be implemented by any concrete child class. It cannot be instantiated on its own.

Program:

```
from abc import ABC, abstractmethod


# Abstract Base Class
# A class that inherits from ABC is an abstract class.
class Shape(ABC):
    # This is an abstract method. Child classes MUST provide an implementation for it.
    @abstractmethod
    def area(self):
        pass



# Concrete Class
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    # Provides the concrete implementation for the `area` method.
    def area(self):
        return 3.14159 * self.radius * self.radius


# Concrete Class
class Rectangle(Shape):
    def __init__(self, width, height):
```

```
        self.width = width
        self.height = height

    # Provides the concrete implementation for the `area` method.
    def area(self):
        return self.width * self.height

# Create objects of the concrete classes
circle_obj = Circle(7)
rectangle_obj = Rectangle(5, 8)

print(f"Area of the circle: {circle_obj.area()}")
print(f"Area of the rectangle: {rectangle_obj.area()}")

# Attempting to create an object of the abstract class will fail.
try:
    abstract_shape = Shape()
except TypeError as e:
    print(f"\nError: {e}")
```

Output:

```
Area of the circle: 153.93791
Area of the rectangle: 40

Error: Can't instantiate abstract class Shape with abstract method area
```