Object-Oriented Programming in Python

Object-Oriented Programming (OOP) is a paradigm that allows the modeling of real-world entities as objects. In Python, OOP principles such as classes, objects, methods, inheritance, and polymorphism are used to design reusable and modular code.

# 1. Class

A class in Python is a blueprint for creating objects. A class can have attributes (variables) and behaviors (methods).

Example 1:

class Animal:  
 def \_\_init\_\_(self, name, species):  
 self.name = name  
 self.species = species  
  
 def make\_sound(self):  
 print(f'{self.name} makes a sound')  
  
# Usage  
dog = Animal('Dog', 'Canine')  
dog.make\_sound() # Output: Dog makes a sound

# 2. Object

An object is an instance of a class. Objects can hold data (attributes) and perform operations (methods).

Example 2:

class Car:  
 def \_\_init\_\_(self, brand, model):  
 self.brand = brand  
 self.model = model  
  
 def details(self):  
 print(f'Car: {self.brand} {self.model}')  
  
# Usage  
car1 = Car('Toyota', 'Corolla')  
car2 = Car('Honda', 'Civic')  
car1.details() # Output: Car: Toyota Corolla  
car2.details() # Output: Car: Honda Civic

# 3. Constructor

The constructor `\_\_init\_\_()` method in Python is called automatically when an object is created. It is used to initialize the object’s attributes.

Example 3:

class Book:  
 def \_\_init\_\_(self, title, author):  
 self.title = title  
 self.author = author  
  
# Usage  
book1 = Book('1984', 'George Orwell')  
print(book1.title) # Output: 1984

# 4. Variables

Variables inside a class are called attributes. These can be instance variables (unique to each object) or class variables (shared across objects).

Example 4:

class Student:  
 school\_name = 'ABC School' # class variable  
  
 def \_\_init\_\_(self, name, grade):  
 self.name = name # instance variable  
 self.grade = grade # instance variable  
  
# Usage  
student1 = Student('John', '5th')  
print(student1.name) # Output: John  
print(Student.school\_name) # Output: ABC School

# 5. Method

A method is a function defined inside a class. It operates on the object's attributes.

Example 5:

class Circle:  
 def \_\_init\_\_(self, radius):  
 self.radius = radius  
  
 def area(self):  
 return 3.14 \* self.radius \*\* 2  
  
# Usage  
circle = Circle(5)  
print(circle.area()) # Output: 78.5

# 6. Dunder (Magic) Methods

Dunder methods are special methods in Python with double underscores. Examples include `\_\_init\_\_()`, `\_\_str\_\_()`, `\_\_len\_\_()`.

Example 6:

class Point:  
 def \_\_init\_\_(self, x, y):  
 self.x = x  
 self.y = y  
  
 def \_\_str\_\_(self):  
 return f'Point({self.x}, {self.y})'  
  
# Usage  
p1 = Point(3, 4)  
print(p1) # Output: Point(3, 4)

# 7. Access Specifiers

Python uses \_ (single underscore) to indicate 'protected' variables, and \_\_ (double underscore) for 'private' variables.

Example 7:

class Employee:  
 def \_\_init\_\_(self, name, salary):  
 self.name = name  
 self.\_salary = salary # protected  
  
 def \_\_display(self): # private  
 print(f'Employee: {self.name}, Salary: {self.\_salary}')  
  
# Usage  
emp = Employee('Alice', 50000)  
emp.\_salary # Can be accessed  
# emp.\_\_display() # Raises AttributeError

# 8. Abstraction

Abstraction hides internal implementation and only shows the required details. Python achieves this through abstract base classes and modules.

Example 8:

from abc import ABC, abstractmethod  
  
class Shape(ABC):  
 @abstractmethod  
 def area(self):  
 pass  
  
class Square(Shape):  
 def \_\_init\_\_(self, side):  
 self.side = side  
  
 def area(self):  
 return self.side \*\* 2  
  
# Usage  
square = Square(4)  
print(square.area()) # Output: 16

# 9. Encapsulation

Encapsulation is the mechanism of wrapping data (attributes) and methods together as a single unit.

Example 9:

class BankAccount:  
 def \_\_init\_\_(self, balance):  
 self.\_\_balance = balance # private  
  
 def deposit(self, amount):  
 self.\_\_balance += amount  
  
 def get\_balance(self):  
 return self.\_\_balance  
  
# Usage  
account = BankAccount(1000)  
account.deposit(500)  
print(account.get\_balance()) # Output: 1500

# 10. Polymorphism

Polymorphism allows different classes to use the same method name, with each class having its own implementation.

Example 10:

class Cat:  
 def sound(self):  
 return 'Meow'  
  
class Dog:  
 def sound(self):  
 return 'Bark'  
  
# Usage  
animals = [Cat(), Dog()]  
for animal in animals:  
 print(animal.sound()) # Output: Meow Bark

# 11. Inheritance

Inheritance allows a class to inherit properties and behaviors from a parent class. It promotes code reusability.

Example 11:

class Vehicle:  
 def \_\_init\_\_(self, brand):  
 self.brand = brand  
  
 def start(self):  
 print(f'{self.brand} vehicle starting')  
  
class Car(Vehicle):  
 def drive(self):  
 print(f'{self.brand} car driving')  
  
# Usage  
car = Car('Tesla')  
car.start() # Output: Tesla vehicle starting  
car.drive() # Output: Tesla car driving

# Assignment

1. Create a class `Laptop` with attributes `brand`, `model`, and `price`. Write methods to get the laptop's details and to update the price.

2. Create a class `Shape` with an abstract method `area()`. Implement two classes `Rectangle` and `Circle` that inherit from `Shape` and calculate the area.

3. Create a class `BankAccount` that encapsulates the balance of a user. Provide methods to deposit, withdraw, and check the balance.

4. Implement a `Person` class with private attributes and methods. Demonstrate the usage of protected and private access specifiers.

# Quiz

1. What is a class in Python?

2. How do you define a constructor in Python?

3. What is the difference between a method and a function?

4. What is polymorphism in OOP?

5. How is inheritance implemented in Python?

6. What is encapsulation?

7. How can you achieve abstraction in Python?

8. What is the purpose of `\_\_init\_\_()` method?

9. What are access specifiers and how are they implemented in Python?

10. What is the use of dunder methods in Python?