Python OOP Basics

Object-Oriented Programming

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data and code. The data is in the form of fields (often known as attributes or properties), and the code is in the form of procedures (often known as methods).

Encapsulation

Encapsulation is one of the fundamental concepts of object-oriented programming, which helps to protect the data and methods of an object from unauthorized access and modification. It is a way to achieve data abstraction, which means that the implementation details of an object are hidden from the outside world, and only the essential information is exposed.

In Python, encapsulation can be achieved by using access modifiers. Access modifiers are keywords that define the accessibility of attributes and methods in a class. The three access modifiers available in Python are public, private, and protected. However, Python does not have an explicit way of defining access modifiers like some other programming languages such as Java and C++. Instead, it uses a convention of using underscore prefixes to indicate the access level.

In the given code example, the class MyClass has two attributes, _protected_var and __private_var. The _protected_var is marked as protected by using a single underscore prefix. This means that the attribute can be accessed within the class and its subclasses but not outside the class. The __private_var is marked as private by using two underscore prefixes. This means that the attribute can only be accessed within the class and not outside the class, not even in its subclasses.

When we create an object of the MyClass class, we can access the _protected_var attribute using the object name with a single underscore prefix. However, we cannot access the __private_var attribute using the object name, as it is hidden from the outside world. If we try to access the __private_var attribute, we will get an AttributeError as shown in the code.

In summary, encapsulation is an important concept in object-oriented programming that helps to protect the implementation details of an object. In Python, we can achieve encapsulation by using access modifiers and using underscore prefixes to indicate the access level.

```
# Define a class named MyClass
class MyClass:

    # Constructor method that initializes the class object
    def __init__(self):

        # Define a protected variable with an initial value of 10
        # The variable name starts with a single underscore, which in
        self._protected_var = 10

        # Define a private variable with an initial value of 20
        # The variable name starts with two underscores, which indica
        self.__private_var = 20

# Create an object of MyClass class
obj = MyClass()

# Access the protected variable using the object name and print its v
# The protected variable can be accessed outside the class but
# it is intended to be used within the class or its subclasses
print(obj._protected_var)   # output: 10

# Try to access the private variable using the object name and print
# The private variable cannot be accessed outside the class, even by
# This will raise an AttributeError because the variable is not acces
print(obj.__private_var)    # AttributeError: 'MyClass' object has no
```

Inheritance

Inheritance promotes code reuse and allows you to create a hierarchy of classes that share common attributes and methods. It helps in creating clean and organized code by keeping related functionality in one place and promoting the concept of modularity. The base class from which a new class is derived is also known as a parent class, and the new class is known as the child class or subclass.

In the code, we define a class named Animal which has a constructor method that initializes the class object with a name attribute and a method named speak. The speak method is defined in the Animal class but does not have a body.

We then define two subclasses named Dog and Cat which inherit from the Animal class. These subclasses override the speak method of the Animal class.

We create a Dog object with a name attribute “Rover” and a Cat object with a name attribute “Whiskers”. We call the speak method of the Dog object using dog.speak(), and it prints “Woof!” because the speak method of the Dog class overrides the speak method of the Animal class. Similarly, we call the speak method of the Cat object using cat.speak(), and it prints “Meow!” because the speak method of the Cat class overrides the speak method of the Animal class.

```
# Define a class named Animal
class Animal:

    # Constructor method that initializes the class object with a nam
    def __init__(self, name):
        self.name = name

    # Method that is defined in the Animal class but does not have a
    # This method will be overridden in the subclasses of Animal
    def speak(self):
        print("")

# Define a subclass named Dog that inherits from the Animal class
class Dog(Animal):

    # Override the speak method of the Animal class
    def speak(self):
        print("Woof!")

# Define a subclass named Cat that inherits from the Animal class
```

```
class Cat(Animal):

    # Override the speak method of the Animal class
    def speak(self):
        print("Meow!")

# Create a Dog object with a name attribute "Rover"
dog = Dog("Rover")

# Create a Cat object with a name attribute "Whiskers"
cat = Cat("Whiskers")

# Call the speak method of the Dog class and print the output
# The speak method of the Dog class overrides the speak method of the
# Therefore, when we call the speak method of the Dog object, it will
dog.speak()   # output: Woof!

# Call the speak method of the Cat class and print the output
# The speak method of the Cat class overrides the speak method of the
# Therefore, when we call the speak method of the Cat object, it will
cat.speak()   # output: Meow!
```

Polymorphism

Polymorphism is an important concept in object-oriented programming that allows you to write code that can work with objects of different classes in a uniform way. In Python, polymorphism is achieved by using method overriding or method overloading.

Method overriding is when a subclass provides its own implementation of a method that is already defined in its parent class. This allows the subclass to modify the behavior of the method without changing its name or signature.

Method overloading is when multiple methods have the same name but different parameters. Python does not support method overloading directly, but it can be achieved using default arguments or variable-length arguments.

Polymorphism makes it easier to write flexible and reusable code. It allows you to write code that can work with different objects without needing to know their specific types.

```python
#The Shape class is defined with an abstract area method, which is in
class Shape:
    def area(self):
        pass

class Rectangle(Shape):
    # The Rectangle class is defined with an __init__ method that ini
    # width and height instance variables.
    # It also defines an area method that calculates and returns
    # the area of a rectangle using the width and height instance var
    def __init__(self, width, height):
        self.width = width  # Initialize width instance variable
        self.height = height  # Initialize height instance variable

    def area(self):
        return self.width * self.height  # Return area of rectangle


 # The Circle class is defined with an __init__ method
 # that initializes a radius instance variable.
 # It also defines an area method that calculates and
 # returns the area of a circle using the radius instance variable.
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius  # Initialize radius instance variable

    def area(self):
        return 3.14 * self.radius ** 2  # Return area of circle using

# The shapes list is created with one Rectangle object and one Circle
# loop iterates over each object in the list and calls the area metho
# The output will be the area of the rectangle (20) and the area of t
shapes = [Rectangle(4, 5), Circle(7)]  # Create a list of Shape objec
for shape in shapes:
    print(shape.area())  # Output the area of each Shape object
```

Abstraction

Abstraction is an important concept in object-oriented programming (OOP) because it allows you to focus on the essential features of an object or system while ignoring the details that aren't relevant to the current context. By reducing complexity and hiding unnecessary details, abstraction can make code more modular, easier to read, and easier to maintain.

In Python, abstraction can be achieved by using abstract classes or interfaces. An abstract class is a class that cannot be instantiated directly, but is meant to be subclassed by other classes. It often includes abstract methods that have no implementation, but provide a template for how the subclass should be implemented. This allows the programmer to define a common interface for a group of related classes, while still allowing each class to have its own specific behavior.

An interface, on the other hand, is a collection of method signatures that a class must implement in order to be considered "compatible" with the interface. Interfaces are often used to define a common set of methods that multiple classes can implement, allowing them to be used interchangeably in certain contexts.

Python does not have built-in support for abstract classes or interfaces, but they can be implemented using the abc (abstract base class) module. This module provides the ABC class and the abstractmethod decorator, which can be used to define abstract classes and methods.

Overall, abstraction is a powerful tool for managing complexity and improving code quality in object-oriented programming, and Python provides a range of options for achieving abstraction in your code.

```
# Import the abc module to define abstract classes and methods
from abc import ABC, abstractmethod
```



```
        pass

# Define a Rectangle class that inherits from Shape
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height
```

```
    # Implement the area method for Rectangles
    def area(self):
        return self.width * self.height

# Define a Circle class that also inherits from Shape
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    # Implement the area method for Circles
    def area(self):
        return 3.14 * self.radius ** 2

# Create a list of shapes that includes both Rectangles and Circles
shapes = [Rectangle(4, 5), Circle(7)]

# Loop through each shape in the list and print its area
for shape in shapes:
    print(shape.area())
```

These are some of the basic OOP principles in Python. This page is currently in progress and more detailed examples and explanations will be coming soon.

Previous page
Context manager

Next page
Dataclasses