Object and Classes

Inheritance and private variables

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class (called a subclass or derived class) to inherit the properties and behaviors of another class (called a superclass or base class). This enables code reuse and promotes a hierarchical structure in your code.

In Python, you can create a subclass by inheriting from a superclass. The syntax for creating a subclass is as follows:

```
class Superclass:
    # Superclass attributes and methods


class Subclass(Superclass):
    # Subclass attributes and methods
```

Here's a simple example to illustrate inheritance:

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


    def speak(self):
        pass


# Subclass
class Dog(Animal):
    def speak(self):
        return "Woof!"


# Subclass
class Cat(Animal):
    def speak(self):
        return "Meow!"


# Creating instances
dog = Dog("Buddy")
cat = Cat("Whiskers")


# Using inherited methods
print(dog.name)      # Output: Buddy
print(dog.speak())   # Output: Woof!


print(cat.name)      # Output: Whiskers
print(cat.speak())   # Output: Meow!
```

In this example, **`Dog`** and **`Cat`** are subclasses of the **`Animal`** superclass. They inherit the **`name`** attribute and have their own implementation of the **`speak`** method. When you create instances of **`Dog`** and **`Cat`**, you can access both the attributes and methods of the superclass.

Inheritance also supports the concept of method overriding, where a subclass can provide a specific implementation for a method defined in the superclass. In the example above, both **`Dog`** and **`Cat`** override the **`speak`** method.

Private Variables

In Python, you can define private variables within a class using a naming convention that indicates their privacy. By convention, variables that are intended to be private are prefixed with a double underscore (**`__`**). These variables are then not accessible directly from outside the class. However, it's important to note that Python does not provide true "private" variables, and their access is still possible through name mangling.

Here's an example:

```
class MyClass:
    def __init__(self):
        # Public variable
        self.public_variable = "I am public!"
        # Private variable
        self.__private_variable = "I am private!"

    def get_private_variable(self):
        return self.__private_variable

    def set_private_variable(self, new_value):
        self.__private_variable = new_value

# Creating an instance of MyClass
obj = MyClass()

# Accessing public variable
print(obj.public_variable)  # Output: I am public!

# Accessing private variable directly (Note: This is discouraged)
# print(obj.__private_variable)  # This would raise an AttributeError

# Accessing private variable through methods
print(obj.get_private_variable())  # Output: I am private!

# Modifying private variable through methods
obj.set_private_variable("New private value")
print(obj.get_private_variable())  # Output: New private value
```

In this example, **`public_variable`** is accessible directly, but **`__private_variable`** is intended to be private. You can still access it using name mangling (e.g., **`_MyClass__private_variable`**), but doing so is discouraged as it goes against the principle of encapsulation.

It's a convention in Python to use a single leading underscore (_) for protected variables (variables that should not be accessed outside the class or its subclasses but are technically still accessible), and a double leading underscore (__) for private variables.