OOP in Python

Part III - Review with Examples

Example - Banking Account

We will learning object-oriented programming (OOP) concepts such as polymorphism, encapsulation, inheritance, and more, through a banking account system example.

Base Account Class

First, we define a Account class that serves as the base for all types of accounts. This class encapsulates the common properties and functionalities of an account.

Encapsulation: The account's balance is made private (using __ prefix), preventing direct access from outside. Instead, balance can only be modified through methods like deposit and withdraw.

Inheritance: This class serves as the base for other specialized account classes (e.g., Checking Account, Savings Account).

Base Account Class

```
class Account:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.__balance = balance  # Encapsulation: balance is private

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"{amount} has been deposited. New balance:{self.__balance}")
        else:
            print("Amount must be positive.")

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"{amount} has been withdrawn. New balance:{self.__balance}")
        else:
            print("Withdrawal amount exceeds the balance or is invalid.")

    def get_balance(self):  # Method to check balance
        return self.__balance
```

Inheritance and Polymorphism

Creating Various Account Types through Inheritance

By inheriting the Account class, we can create specific types of accounts that have unique conditions or interest rates, such as SavingsAccount and CheckingAccount

Utilizing Polymorphism

Polymorphism is an OOP feature that allows objects of different classes to respond to the same message (method call) in different ways. In the example above, the CheckingAccount class overrides the withdraw method from the Account class to implement transaction fees specific to checking accounts.

Subclasses

```
class SavingsAccount(Account):  # Savings account
    def __init__(self, owner, balance=0, interest_rate=0.01):
        super().__init__(owner, balance)
        self.interest_rate = interest_rate

    def apply_interest(self):  # Apply interest
        interest = self.get_balance() * self.interest_rate
        self.deposit(interest)
        print(f"Interest {interest} has been applied.")

class CheckingAccount(Account):  # Checking account
    def __init__(self, owner, balance=0, transaction_fee=1.00):
        super().__init__(owner, balance)
        self.transaction_fee = transaction_fee

    def withdraw(self, amount):  # Custom withdraw method for checking account
        if amount + self.transaction_fee <= self.get_balance():
            super().withdraw(amount+self.transaction_fee)  # Call the original withdraw method
            print(f"Transaction fee of {self.transaction_fee} has been applied.")
        else:
            print("Withdrawal amount including fees exceeds balance.")
```

Example Usage

```
acc = Account("John Doe", 1000)
acc.deposit(500)
acc.withdraw(200)

savings = SavingsAccount("Jane Doe", 1000, 0.05)
savings.apply_interest()

checking = CheckingAccount("Alex Smith", 1000, 2.00)
checking.withdraw(100)  # Withdrawal in checking account includes transaction fee.
```

Results

```
500 has been deposited. New balance: 1500

200 has been withdrawn. New balance: 1300

50.0 has been deposited. New balance: 1050.0

Interest 50.0 has been applied.

102.0 has been withdrawn. New balance: 898.0

Transaction fee of 2.0 has been applied.
```

Hidden/Private Variable

Hidden Variable

```
class Car:
    # Hidden member of Car
    __mileage = 0

    # A member method that changes __mileage
    def drive(self, miles):
        self.__mileage += miles
        print(f"Driven {miles} miles. Total mileage is now {self.__mileage} miles.")

# Driver code
myCar = Car()
myCar.drive(50)
myCar.drive(100)

# This line attempts to access the hidden variable directly and will cause an error
# print(myCar.__mileage)
```

Hidden Variable

Class Car: Defines a car with a hidden or private variable __mileage. This variable is meant to represent the total miles driven by the car and is not directly accessible from outside the class.

Method drive(miles): A public method that simulates driving the car a certain number of miles and increments the __mileage variable accordingly. It also prints the mileage after each drive.

Driver Code: Creates an instance of Car named myCar and simulates driving by calling myCar.drive(50) and then myCar.drive(100), which updates and prints the mileage each time.

Accessing Hidden Variable: The commented-out line # print(myCar.__mileage) demonstrates what happens if you try to access the hidden variable directly. This will raise an AttributeError because __mileage is private to the Car class and not directly accessible from outside the class.

Hidden Variable

```
class Car:
    # Hidden member of Car
    __mileage = 0

    # A member method that changes __mileage
    def drive(self, miles):
        self.__mileage += miles
        print(f"Driven {miles} miles. Total mileage is now {self.__mileage} miles.")

# Driver code
myCar = Car()
myCar.drive(50)
myCar.drive(100)

print(myCar._Car__mileage)
```

`__str__` and `__repr__`

In Python, __str__ and __repr__ are special methods that define string representations of objects. While these methods serve similar purposes, they are typically used in different contexts and have different goals.

Example

```
class Person:

    def __init__(self, name, age):

        self.name = name

        self.age = age


    def __str__(self):

        return f"{self.name} is {self.age} years old."


    def __repr__(self):

        return f"Person('{self.name}', {self.age})"
```

__str__

The __str__ method is used to provide a human-readable representation of an object, primarily for display purposes.

This method is automatically invoked by the print() function and when using the str() built-in function to convert the object to a string.

If a class does not define the __str__ method, Python will default to using the object's __repr__ method as a fallback for its string representation.

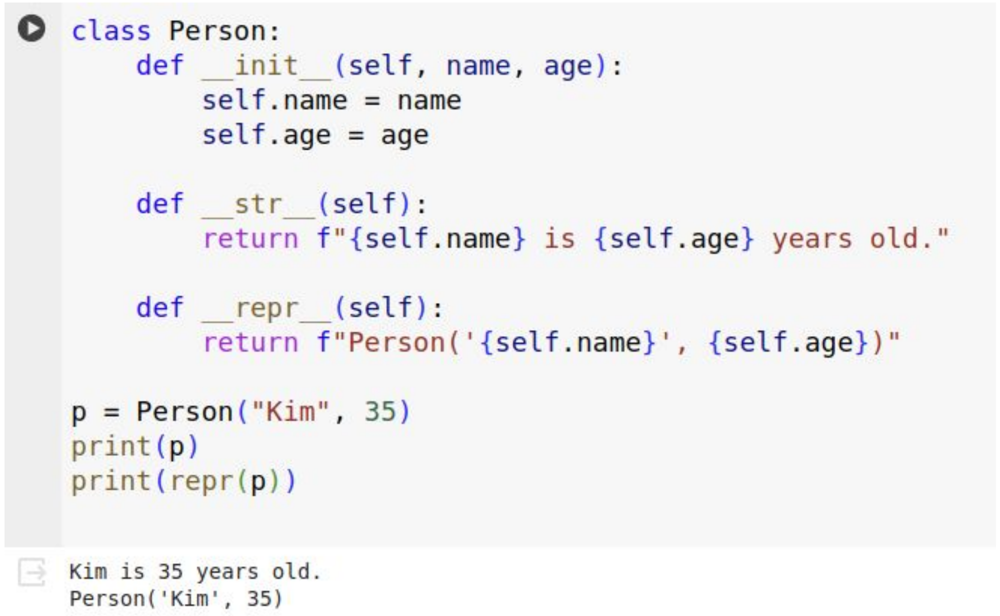
__repr__

The __repr__ method is intended to provide an unambiguous representation of an object, aimed at developers. It's used for debugging and logging purposes.

The goal of __repr__ is to return a string that, when passed to eval(), could (in theory) produce an object with the same properties as the original object. In other words, its result should be a valid Python expression when possible.

All Python objects come with a built-in implementation of __repr__, and if not explicitly overridden in a class, it returns the default representation that includes the object’s address in memory.

Example



```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name} is {self.age} years old."

    def __repr__(self):
        return f"Person('{self.name}', {self.age})"

p = Person("Kim", 35)
print(p)
print(repr(p))
```

```
Kim is 35 years old.
Person('Kim', 35)
```

In this example, `__str__` focuses on providing a user-friendly description of the object, suitable for end-user consumption, whereas `__repr__` aims at giving a more precise and formal representation of the object, potentially allowing for the recreation of the original object from its string representation.

Multiple Inheritance

Multiple Inheritance

```
class Engine():
    def __init__(self):
        self.power = "120hp"
        print("Engine is ready")

class Body():
    def __init__(self):
        self.type = "Sedan"
        print("Body is ready")

class Car(Engine, Body):
    def __init__(self):
        # Calling constructors of Engine and Body classes
        Engine.__init__(self)
        Body.__init__(self)
        print("Car is ready")

    def specifications(self):
        print(f"Power: {self.power}, Body Type: {self.type}")

my_car = Car()
my_car.specifications()
```

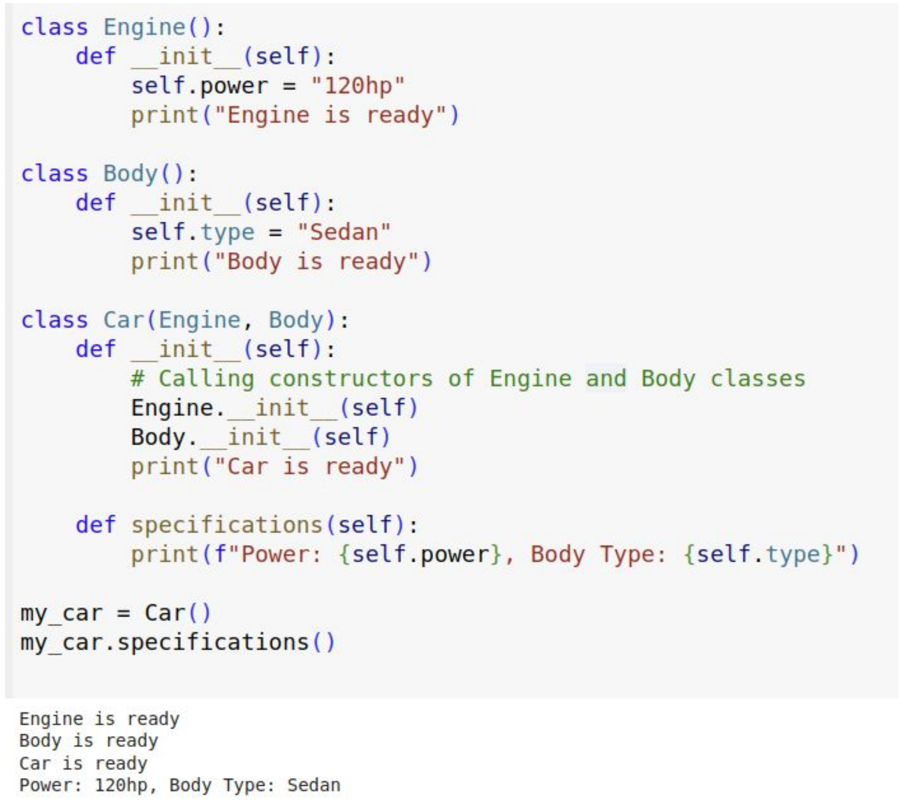
Multiple Inheritance

Car Class: Inherits from both Engine and Body. In its constructor, it explicitly calls the constructors of both Engine and Body to ensure that the car is equipped with both engine power and body type. After initializing its components, it prints "Car is ready".

The Car class demonstrates multiple inheritance by inheriting properties (power and type) and initializing behavior from both Engine and Body classes.

By explicitly calling the constructors of the Engine and Body classes within its constructor, the Car class ensures that all necessary initializations for its components are performed.

Multiple Inheritance



```python
class Engine():
    def __init__(self):
        self.power = "120hp"
        print("Engine is ready")

class Body():
    def __init__(self):
        self.type = "Sedan"
        print("Body is ready")

class Car(Engine, Body):
    def __init__(self):
        # Calling constructors of Engine and Body classes
        Engine.__init__(self)
        Body.__init__(self)
        print("Car is ready")

    def specifications(self):
        print(f"Power: {self.power}, Body Type: {self.type}")

my_car = Car()
my_car.specifications()
```

```
Engine is ready
Body is ready
Car is ready
Power: 120hp, Body Type: Sedan
```

__call__

In Python, the __call__ method is a special method that allows an instance of a class to be called as if it were a function.

Essentially, if a class defines a __call__ method, it can make its instances callable, just like a function. This can be particularly useful when you want your objects to behave like functions, or when you want to use classes to define objects that need to be invoked for a specific purpose.

__call__

```
class SquareCalculator:

    def __call__(self, x):

        return x * x


# Creating an instance of SquareCalculator

calculator = SquareCalculator()


# Using the instance as if it were a function

result = calculator(5)  # This calls the __call__ method

print(result)  # Output: 25
```

In this example, the `SquareCalculator` class has a `__call__` method that takes a single argument `x` and returns its square.

When we create an instance of `SquareCalculator` named `calculator`, we can "call" `calculator` with a number as if it were a function, and it returns the square of that number.

Why Use __call__

Using the __call__ method can make your code more intuitive and elegant, especially when the object's main purpose is to perform a specific operation or calculation. It can also be used to maintain state or configuration that affects its behavior when called.