Training report day – 15

22 June 2024

Inheritance in Python:

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows one class (child or subclass) to inherit the properties and methods of another class (parent or superclass). It promotes code reusability by enabling a new class to take on the attributes and behaviors of an existing class.

Key concepts:

1. Parent class(Super class):
    - Also known as a base class or super class.
    - It’s the class whose attributes and methods are inherited by another class.
2. Child class:
    - Also known as a derived class or sub class.
    - It inherites attributes and methods from ites parent class .
3. Types of inheritance:
    - Single Inheritance: A child class inherits from only one parent class.
    - Multiple Inheritance: A child class inherits from multiple parent classes.
    - Multilevel Inheritance: One class is derived from another, which is itself derived from another class.
    - Hierarchical Inheritance: Multiple child classes inherit from the same parent class.
    - Hybrid Inheritance: Combination of two or more types of inheritance.

Advantages of Inheritance:

- Code Reusability: Avoids redundant code by inheriting from existing classes.
- Modularity: Promotes a modular approach to software development.
- Ease of Maintenance: Changes made in the parent class automatically reflect in all child classes (depending on the design).

Single inheritance:

Example:

```
# Parent class
class Animal:
    def __init__(self, species, legs):
        self.species = species
        self.legs = legs
    def make_sound(self):
        return "Some generic sound"

# Child class inheriting from Animal
class Dog(Animal):
    def __init__(self, species, legs, breed):
        super().__init__(species, legs)
        self.breed = breed

    def make_sound(self):
        return "Woof!"

    def describe(self):
        return f"A {self.breed} dog ({self.species}) with {self.legs} legs"

# Usage
my_dog = Dog("Canine", 4, "Labrador")
print(my_dog.make_sound())
print(my_dog.describe())
```

Multiple inheritance:

```
# Parent class 1
class Father:
    def __init__(self, eye_color):
        self.eye_color = eye_color

    def play_game(self):
        return "Playing chess with dad"

# Parent class 2
class Mother:
    def __init__(self, hair_color):
        self.hair_color = hair_color

    def cooking(self):
        return "Cooking with mom"

# Child class inheriting from both Father and Mother
class Child(Father, Mother):
    def __init__(self, eye_color, hair_color, name):
        Father.__init__(self, eye_color)
        Mother.__init__(self, hair_color)
        self.name = name
    def play(self):
        return f"{self.name} likes {super().play_game()} and
{super().cooking()}"

# Usage
my_child = Child("Blue", "Brown", "Emma")
print(my_child.play())
```

Multilevel inheritance:

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

    def greet(self):
        return f"Hello, {self.name}!"

# Derived class inheriting from Base
class Derived(Base):
    def __init__(self, name, age):
        super().__init__(name)
        self.age = age

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

# FurtherDerived class inheriting from Derived
class FurtherDerived(Derived):
    def __init__(self, name, age, hobby):
        super().__init__(name, age)
        self.hobby = hobby

    def show_hobby(self):
        return f"{self.name}'s hobby is {self.hobby}"

# Usage
person = FurtherDerived("Alice", 30, "Painting")
print(person.greet())
print(person.describe())
print(person.show_hobby())
```

Hierarchical inheritance:

```
# Base class
class Shape:
    def __init__(self, color):
        self.color = color

    def area(self):
        pass

# Derived classes inheriting from Shape
class Rectangle(Shape):
    def __init__(self, color, width, height):
        super().__init__(color)
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

class Circle(Shape):
    def __init__(self, color, radius):
        super().__init__(color)
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

class Triangle(Shape):
    def __init__(self, color, base, height):
        super().__init__(color)
        self.base = base
        self.height = height

    def area(self):
        return 0.5 * self.base * self.height

# Usage
rectangle = Rectangle("Red", 5, 10)
circle = Circle("Blue", 7)
triangle = Triangle("Green", 4, 6)

print(rectangle.area())
print(circle.area())
print(triangle.area())
```

Hybrid inheritance:

```
# Base class
class LivingBeing:
    def __init__(self, kingdom):
```

```python
        self.kingdom = kingdom

    def breathe(self):
        return "Breathing..."

# Intermediate base class
class Animal(LivingBeing):
    def __init__(self, kingdom, habitat):
        super().__init__(kingdom)
        self.habitat = habitat

    def sound(self):
        pass

# Another intermediate base class
class Mammal(Animal):
    def __init__(self, kingdom, habitat, warm_blooded=True):
        super().__init__(kingdom, habitat)
        self.warm_blooded = warm_blooded

    def has_fur(self):
        return "Has fur or hair"

# Derived class
class Human(Mammal):
    def __init__(self, kingdom, habitat, name):
        super().__init__(kingdom, habitat)
        self.name = name

    def sound(self):
        return "Speaking..."

    def walk(self):
        return "Walking on two legs"

# Usage
human = Human("Animalia", "Land", "Alice")
print(human.breathe())      # Output: Breathing...
print(human.sound())        # Output: Speaking...
print(human.has_fur())      # Output: Has fur or hair
print(human.walk())         # Output: Walking on two legs
```

Encapsulation in Python:

Encapsulation is the concept of Object Oriented Programming. It avoids the accidental change in any variable. The value or content of the variable can be accessed by any method of the object.

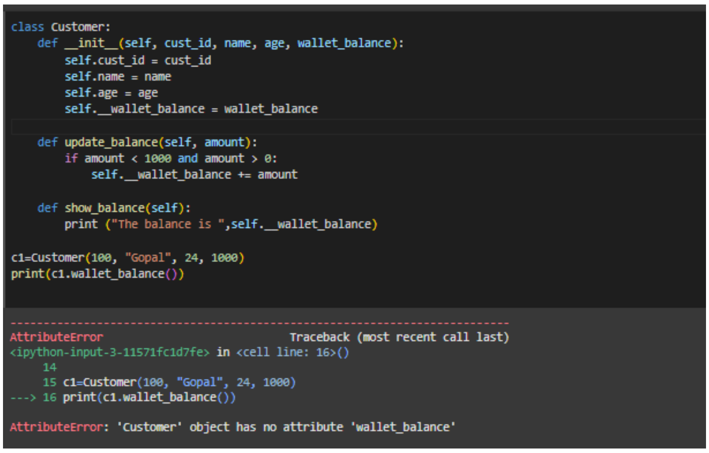
Public data access:

Public data can be accessed by the other functions and can be changed by any outer method. This can change any critical value in the code.

Private data access:

We can put a lock on that data by adding a double underscore in front of it, as shown in below code.

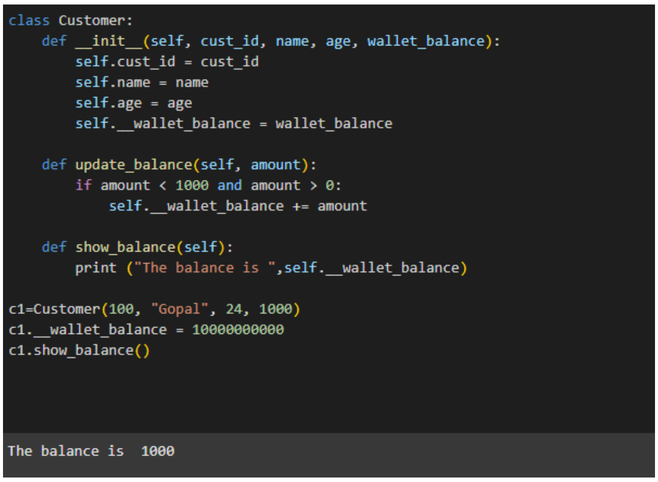
Adding a double underscore makes the attribute a private attribute. Private attributes are those which are accessible only inside the class. This method of restricting access to our data is called encapsulation.



```
class Customer:
    def __init__(self, cust_id, name, age, wallet_balance):
        self.cust_id = cust_id
        self.name = name
        self.age = age
        self.__wallet_balance = wallet_balance

    def update_balance(self, amount):
        if amount < 1000 and amount > 0:
            self.__wallet_balance += amount

    def show_balance(self):
        print ("The balance is ",self.__wallet_balance)

c1=Customer(100, "Gopal", 24, 1000)
print(c1.wallet_balance())
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-3-11571fc1d7fe> in <cell line: 16>()
     14
     15 c1=Customer(100, "Gopal", 24, 1000)
---> 16 print(c1.wallet_balance())

AttributeError: 'Customer' object has no attribute 'wallet_balance'
```

Note: Private variable can be accessed in other method of same class but cannot accessed by any method outside the class.



```
class Customer:
    def __init__(self, cust_id, name, age, wallet_balance):
        self.cust_id = cust_id
        self.name = name
        self.age = age
        self.__wallet_balance = wallet_balance

    def update_balance(self, amount):
        if amount < 1000 and amount > 0:
            self.__wallet_balance += amount

    def show_balance(self):
        print ("The balance is ",self.__wallet_balance)

c1=Customer(100, "Gopal", 24, 1000)
c1.__wallet_balance = 10000000000
c1.show_balance()
```

```
The balance is  1000
```