DATE : 15 june 2024

DAY : Saturday

TOPICS : Abstraction, Encapsulation and Inheritance

⌄ **ABSTRACTION**

Consider the below given code:

```python
class Mobile:
    def __init__(self, brand, price):
        print("Inside constructor")
        self.brand = brand
        self.price = price
    def purchase(self):
        print("Purchasing a mobile")
        print("This mobile has brand", self.brand, "and price", self.price)
```

```python
print("Mobile-1")
mob1=Mobile("Apple", 20000)
mob1.purchase()
print("Mobile-2")
mob2=Mobile("Samsung",3000)
mob2.purchase()
```

```
⇥  Mobile-1
    Inside constructor
    Purchasing a mobile
    This mobile has brand Apple and price 20000
    Mobile-2
    Inside constructor
    Purchasing a mobile
    This mobile has brand Samsung and price 3000
```

When we invoke the purchase() on a mobile object, we don't have to know the details of the method to invoke it.

This ability to use something without having to know the details of how it is working is called as **abstraction.**

⌄ Need for Encapsulation

What is the use of a lock?

Consider the below code where the customer has a wallet_balance and there are methods which allow us to access and update that balance based on some logic. **Just the way a lock prevents others from accessing your property, we can restrict other parts of the code from directly accessing sensitive data.**

```python
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.update_balance(500)
c1.show_balance()
```

    The balance is 1500

˅ Public Data Access

But with the way currently it is coded, the data can be accidentally changed by directly assigning a incorrect value to it as shown below:

```python
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  10000000000
```

∨ 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.

```python
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-4-e9efd40e0348> 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'
```

∨ How does encapsulation work?

Problem Statement

When we put a double underscore in front of the attribute name, python will internally change its name to **_Classname__attribute.**

This is why we get an error when we try to access a private attribute.

```python
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-5-d06fe1d9e526> in <cell line: 16>()
     14
     15 c1=Customer(100, "Gopal", 24, 1000)
---> 16 print(c1.__wallet_balance)
     17

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

⌄ **Private Data Update - Caution !**

Problem Statement

If we try to assign a value to a private variable, we end up creating a new attribute in python. Thus this code does not give an error, but it is logically flawed and does not produce the intended result.

```python
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

⌄ **Accessing Private Variables**

Problem Statement

Since we know that the name of the variable changes when we make it private, we can access it using its modified name as shown in below code.

So, if private variable can be accessed outside the class and can be modified, then what is the advantage of making it private?

Note: Languages like Java, C#, etc do not allow access of private variable outside the class

```python
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._Customer__wallet_balance = 10000000000
c1.show_balance()
```

⇥▾   The balance is 10000000000

˅ Getters & Setters

To have a error free way of accessing and updating private variables, we create specific methods for this.

- The methods which are meant to set a value to a private variable are called **setter methods.**

- The methods meant to access private variable values are called **getter methods.**

The below code is an example of getter and setter methods:

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

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

    def get_wallet_balance(self):
        return self.__wallet_balance

c1=Customer(100, "Gopal", 24, 1000)
c1.set_wallet_balance(120)
print(c1.get_wallet_balance())
```

⇥ 120

Class Diagram

A lot of things can go wrong in communication.

To ensure that programmers all over understand each other properly, we need a common way of representing a class. This is called as a class diagram. This is similar to a circuit diagram or a plan or machine diagram which allows engineers to understand each others' ideas clearly.

⌄ Visibility Notation:

Visibility notations indicate the access level of attributes and methods. Common visibility notations include:

- **+** for public (visible to all classes)
- **-** for private (visible only within the class)
- **#** for protected (visible to subclasses)
- **~** for package or default visibility (visible to classes in the same package)
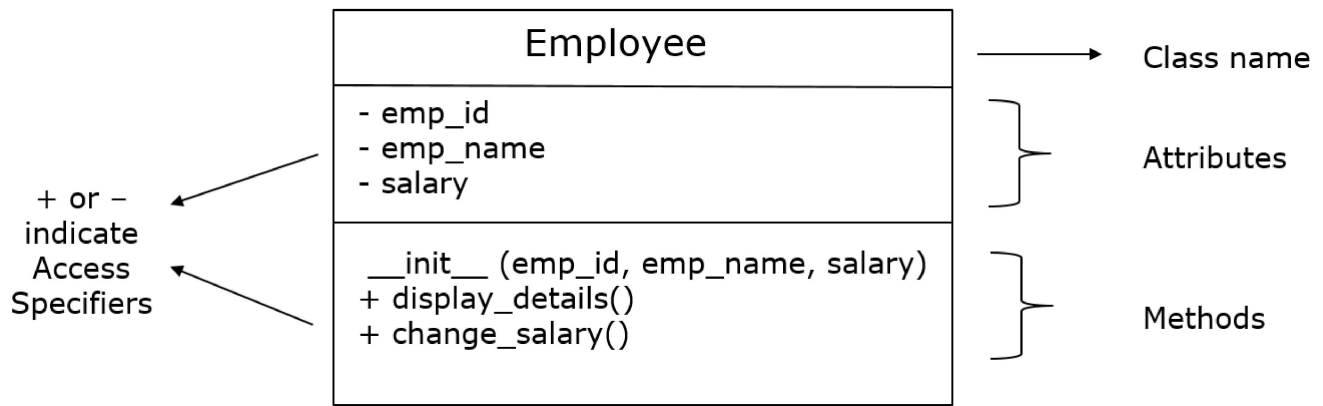
Note: We can create private methods by adding a double underscore in front of it, just like private variables. Also, if a method has both leading and trailing double underscores (like __ init __ , __ str __, etc) it indicates that it is a special built-in method. As per coding convention, we are not supposed to create our own methods with both leading and trailing underscores.

Exercise on Access Specifiers - Level 1

In the Athlete class given below,

- make all the attributes private and
- add the necessary accessor and mutator methods

Represent Maria, the runner and make her run.

```python
class Athlete:
    def __init__(self,name,gender):
        self.name=name
        self.gender=gender
    def running(self):
        if(self.gender=="girl"):
            print("150mtr running")
        else:
            print("200mtr running")
A_1 = Athlete("Maria","girl")
A_1.running()
```

    150mtr running

Getters & Setters - Way to Code !

All setter methods must accept the value to be updated as a parameter and all getter methods must not have any parameter and they must return the value.

Setter methods are called as mutator methods (as they change or mutate the value) and the getter methods are called accessor methods (as they access the values)

```python
class Customer:
    def __init__(self, id, name, age, wallet_balance):
        self.id = id
        self.name = name
        self.age = age
        self.__wallet_balance = wallet_balance
    def set_wallet_balance(self, amount):
        if amount < 1000 and amount>  0:
            self.__wallet_balance = amount
    def get_wallet_balance(self):
        return self.__wallet_balance
c1=Customer(100, "Gopal", 24, 1000)
c1.set_wallet_balance(120)
print(c1.get_wallet_balance())
```

⇥ 120

Abstraction & Encapsulation - Summary

- Encapsulation is preventing access to a data outside the class

- Adding a __ in front of a attribute makes it private

- In python, adding a __ changes the name of the attribute to _Classname__attribute

Introduction to Inheritance

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 (Superclass)**:
   - Also known as a base class or superclass.
   - It's the class whose attributes and methods are inherited by another class.

2. **Child Class (Subclass)**:

- Also known as a derived class or subclass.
- It inherits attributes and methods from its parent class and can also have its own additional attributes and methods.

3. **Inheritance Syntax in Python**:

- In Python, inheritance is declared by specifying the parent class(es) in the definition of a child class.
- Syntax: `class ChildClassName(ParentClassName):`

4. **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).

⌄ Types of Inheritance

Single Inheritance Scenario:

Imagine a scenario where you have a base class Animal with generic attributes and methods, and a derived class Dog that inherits from Animal and adds specific attributes and methods related to dogs.

Single Inheritance Example:

```python
# 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())
```

```
Woof!
A Labrador dog (Canine) with 4 legs
```

Multiple Inheritance Scenario:

Consider a scenario where you have two parent classes, Father and Mother, and a child class Child that inherits from both Father and Mother, thereby inheriting attributes and methods from both parents.

Multiple Inheritance Example:

```python
# 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())
```

> Emma likes Playing chess with dad and Cooking with mom

Multilevel Inheritance Scenario:

In multilevel inheritance, a derived class serves as the base class for another class. For instance, consider a scenario where you have a Base class representing a basic entity, a Derived class inheriting from Base, and a FurtherDerived class inheriting from Derived, adding more specific attributes and methods.

Multilevel Inheritance Example:

```python
# 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())
```

```
Hello, Alice!
Alice is 30 years old
Alice's hobby is Painting
```

Hierarchical Inheritance Scenario:

In hierarchical inheritance, multiple derived classes inherit from a single base class. Consider a scenario where you have a Shape base class and multiple derived classes like Rectangle, Circle, and Triangle, each inheriting from Shape and adding specific attributes and methods.

Hierarchical Inheritance Example:

```python
# 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):
```