Encapsulation

Encapsulation in Python

In Python, encapsulation is a method of wrapping data and functions into a single entity. For example, A class encapsulates all the data (methods and variables). Encapsulation means the internal representation of an object is generally hidden from outside of the object's definition.

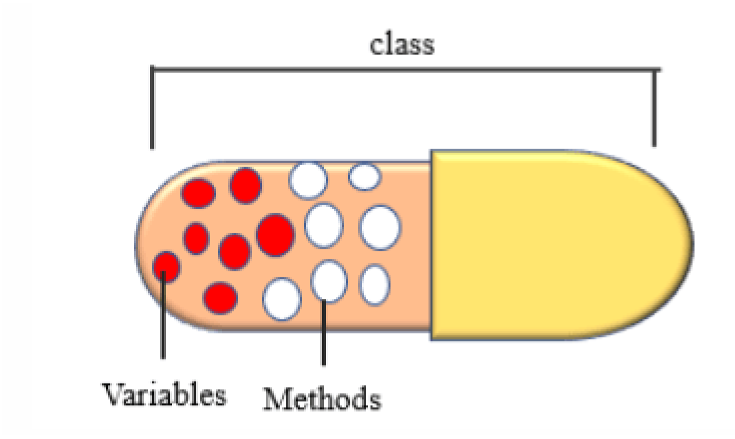

Fig: Python Encapsulation

Need of Encapsulation

Encapsulation acts as a protective layer. We can restrict access to methods and variables from outside, and It can prevent the data from being modified by accidental or unauthorized modification. Encapsulation provides security by hiding the data from the outside world.

Example: Encapsulation in Python

When you create a class, it means you are implementing encapsulation. A class is an example of encapsulation as it binds all the data members (instance variables) and methods into a single unit.

In Python, we do not have access modifiers, such as public, private, and protected. But we can achieve encapsulation by using prefix **single underscore** and **double underscore** to control access of variable and method within the Python program.

```
class Employee:
    def __init__(self, name, salary):
        # public member
        self.name = name
        # private member
        # not accessible outside of a class
        self.__salary = salary

    def show(self):
        print("Name is ", self.name, "and salary is", self.__salary)

emp = Employee("Jessa", 40000)
emp.show()

# access salary from outside of a class
print(emp.__salary)
```

Output:

```
Name is  Jessa and salary is 40000
AttributeError: 'Employee' object has no attribute '__salary'
```

In the above example, we create a class called `Employee`. Within that class, we declare two variables `name` and `__salary`. We can observe that the `name` variable is accessible, but `__salary` is the **private variable**. We cannot access it from outside of class. If we try to access it, we will get an error

Polymorphism

Polymorphism in OOP is the **ability of an object to take many forms**. In simple words, polymorphism allows us to perform the same action in many different ways.

Polymorphism is taken from the Greek words Poly (many) and morphism (forms). Polymorphism defines the ability to take different forms.

For example, The student can act as a student in college, act as a player on the ground, and as a daughter/brother in the home. Another example in the programming language, the + operator, acts as a concatenation and arithmetic addition.

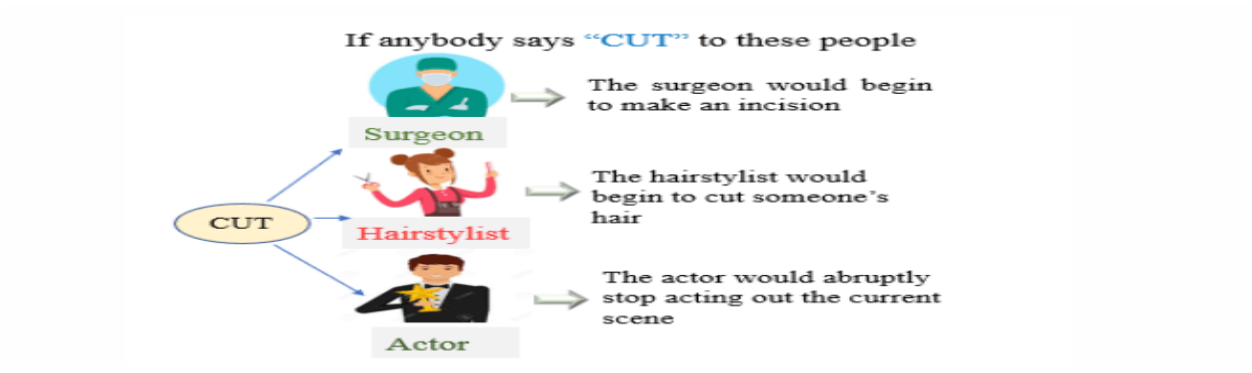
If anybody says "CUT" to these people
Surgeon
The surgeon would begin to make an incision
CUT
Hairstylist
The hairstylist would begin to cut someone's hair
Actor
The actor would abruptly stop acting out the current scene

Fig: Python Polymorphism

Read the complete guide on **Polymorphism in Python**.

Example: Using Polymorphism in Python

For example, In the below example, calculate_area() instance method created in both Circle and Rectangle class. Thus, we can create a function that takes any object and calls the object's calculate_area() method to implement polymorphism. Using this object can perform

Polymorphism with class methods is useful when we want objects to perform the same action in different ways. In the below example, both objects calculate the area (same action) but in a different way (different formulas)

```
class Circle:
    pi = 3.14

    def __init__(self, redius):
        self.radius = redius

    def calculate_area(self):
        print("Area of circle :", self.pi * self.radius * self.radius)

class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def calculate_area(self):
        print("Area of Rectangle :", self.length * self.width)

# function
def area(shape):
    # call action
    shape.calculate_area()

# create object
cir = Circle(5)
rect = Rectangle(10, 5)

# call common function
area(cir)
area(rect)
```

Output:

```
Area of circle : 78.5
Area of Rectangle : 50
```