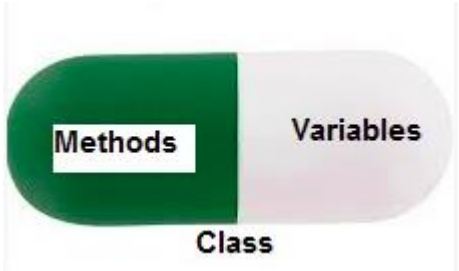
Python OOP: Encapsulation, Polymorphism, and Abstraction

Encapsulation in Python

What is Encapsulation?



- Encapsulation is one of the fundamental concepts in object-oriented programming (OOP).
- It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data.
- To prevent accidental change, an object's variable can only be changed by an object's method. Those types of variables are known as private variables.
- A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.

Protected Variables

- Protected variables are those data members of a class that can be accessed within the class and the classes derived from that class. In Python, there is no existence of “Public” instance variables. However, we use underscore ‘_’ symbol to determine the access control of a data member in a class.
- Any member prefixed with an underscore should be treated as a non-public part of the API or any Python code, whether it is a function, a method or a data member.

Private Members

- Like protected members, the difference is that the class members declared private should neither be accessed outside the class nor by any base class.
- In Python, there is no existence of Private instance variables that cannot be accessed except inside a class.
- However, to define a private member prefix the member's name with double underscore “__”.

- **Note: Python’s private and protected members can be accessed outside the class through python name mangling.**

Private Members Example

```python
class TestClass:
    def __init__(self, a, b):
        self._a = a
        self.__b = b


    def _test(self):
        print("From the test class")


class OtherTestClass(TestClass):
    def _test(self):
        print("From the other test class")
```

```
test._test()
```

```
From the other test class
```

```
test._a
```

```
1
```

```
test.__b
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[57], line 1
----> 1 test.__b

AttributeError: 'OtherTestClass' object has no attribute '__b'
```

Encapsulation Example

```
Calling protected member of base class:  2
Calling modified protected member outside class:  3
Accessing protected member of obj1:  3
Accessing protected member of obj2:  2
```

```python
# Python program to
# demonstrate protected members

# Creating a base class
class Base:
    def __init__(self):

        # Protected member
        self._a = 2

# Creating a derived class
class Derived(Base):
    def __init__(self):

        # Calling constructor of
        # Base class
        Base.__init__(self)
        print("Calling protected member of base class: ",
              self._a)

        # Modify the protected variable:
        self._a = 3
        print("Calling modified protected member outside class: ",
              self._a)


obj1 = Derived()

obj2 = Base()

# Calling protected member
# Can be accessed but should not be done due to convention
print("Accessing protected member of obj1: ", obj1._a)

# Accessing the protected variable outside
print("Accessing protected member of obj2: ", obj2._a)
```

Name Mangling

- In name mangling process any identifier with two leading underscore and one trailing underscore is textually replaced with _classname__identifier where classname is the name of the current class.
- It means that any identifier of the form __geek (at least two leading underscores or at most one trailing underscore) is replaced with _classname__geek, where classname is the current class name with leading underscore(s) stripped.
- Mangling refers to how it modifies the attribute or method by adding the class name to it.

Encapsulation Example

In this example, the class variable __name is not accessible outside the class.

It can be accessed only within the class.

Any modification of the class variable can be done only inside the class.

```
class Student:
        def __init__(self, name):
                self.__name = name

        def displayName(self):
                print(self.__name)

s1 = Student("Santhosh")
s1.displayName()

# Raises an error
print(s1.__name)
```

```
Santhosh
```

```
Traceback (most recent call last):
  File "/home/be691046ea08cd2db075d27186ea0493.py", line 14, in
    print(s1.__name)
AttributeError: 'Student' object has no attribute '__name'
```

Name Mangling Process

With the help of dir() method, we can see the name mangling process that is done to the class variable.

The name mangling process was done by the Interpreter.

The dir() method is used by passing the class object and it will return all valid attributes that belong to that object.

The above output shows dir() method returning all valid attributes with the name mangling process that is done to the class variable __name. The name changed from __name to _Student__name.

```
class Student:
        def __init__(self, name):
                self.__name = name

s1 = Student("Santhosh")
print(dir(s1))
```

```
['_Student__name', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
    '__eq__', '__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__',
    '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__',
    '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
    '__sizeof__', '__str__', '__subclasshook__', '__weakref__']
```

Accessing Name Mangled Variables

The name mangling process helps to access the class variables from outside the class.

The class variables can be accessed by adding _classname to it.

The name mangling is closest to private not exactly private.

The class variable is accessed by adding the _classname to it. The class variable is accessed from outside the class with the name _Student__name.

```
class Student:
        def __init__(self, name):
                self.__name = name

s1 = Student("Santhosh")
print(s1._Student__name)
```

```
Santhosh
```

Note about Mangling

- It's important to note that name mangling in Python is more about convention than about enforcing strict access control.
- Attributes and methods prefixed with double underscores are still technically accessible from outside the class, but the mangling mechanism makes it less likely for accidental access or overriding to occur.
- Mangling is not exactly encapsulation, rather, it’s a support to implement better encapsulation.

Emphasis on Name Mangling to Support Encapsulation

This code defines a class EncapsulationExample with private and protected attributes.

It demonstrates encapsulation by marking attributes as private or protected and uses name mangling to prevent direct access to private attributes from outside the class.

Encapsulation is more on the __init__ part, as the attributes there are only accessible within the class.

```
class EncapsulationExample:
    def __init__(self):
        self.__private_attribute = 42
        self._protected_attribute = 10

    def print_info(self):
        print("Private Attribute:", self.__private_attribute)
        print("Protected Attribute:", self._protected_attribute)


# Example usage
obj = EncapsulationExample()

# Accessing protected attribute directly (not recommended but possible)
print("Protected Attribute (Direct Access):", obj._protected_attribute)

# Trying to access private attribute directly (Name mangling)
# Note: This will not work because the name has been mangled
# print("Private Attribute (Direct Access):", obj.__private_attribute)
```

Getters, Setters, Property

- Getters: These are the methods used in OOPS which helps to access the private attributes from a class.
- Setters: These are the methods used in OOPS feature which helps to set the value to private attributes in a class.

Private Attribute Encapsulation

SampleClass has three methods.

__init__:- It is used to initialize the attributes or properties of a class.

__a:- It is a private attribute.

get_a:- It is used to get the values of private attribute a.

set_a:- It is used to set the value of a using an object of a class.

You are not able to access the private variables directly in Python. That's why you implemented the getter method.

```
class SampleClass:

    def __init__(self, a):
        ## private varibale or property in Python
        self.__a = a

    ## getter method to get the properties using an object
    def get_a(self):
        return self.__a

    ## setter method to change the value 'a' using an object
    def set_a(self, a):
        self.__a = a
## creating an object
obj = SampleClass(10)

## getting the value of 'a' using get_a() method
print(obj.get_a())

## setting a new value to the 'a' using set_a() method
obj.set_a(45)

print(obj.get_a())
```

```
10
45
```

Implement private attributes, getters, and setters in Python.

The same process was followed in Java. Let's write the same implementation in a Pythonic way.

You don't need any getters, setters methods to access or change the attributes. You can access it directly using the name of the attributes.

```
class PythonicWay:

    def __init__(self, a):
        self.a = a
```

```
## Creating an object for the 'PythonicWay' class
obj = PythonicWay(100)

print(obj.a)
```

```
100
```

What's the difference between the two classes?

- SampleClass hides the private attributes and methods. It implements the encapsulation feature of OOPS.
- PythonicWay doesn't hide the data. It doesn't implement any encapsulation feature.

What's the better to use?

- This depends on the need.
- If you want private attributes and methods you can implement the class using setters, getters methods otherwise you will implement using the normal way.

Polymorphism

What is Polymorphism?

- The word polymorphism means having many forms.
- In programming, polymorphism means the same function name (but different signatures) being used for different types.
- The key difference is the data types and number of arguments used in function.

Polymorphism sample code

```python
# A simple Python function to demonstrate
# Polymorphism

def add(x, y, z = 0):
    return x + y+z

# Driver code
print(add(2, 3))
print(add(2, 3, 4))
```

```
5
9
```

Polymorphism sample code

```python
class Bird:
    def fly(self):
        print("Started to fly")
```

```python
class Chicken(Bird):
    def fly(self):
        print("A chicken can't fly")
```

```python
class Penguin(Bird):
    def fly(self):
        print("A Penguin can't fly")
```

```python
def make_bird_fly(bird):
    bird.fly()

bird = Bird()
chicken = Chicken()
penguin = Penguin()

make_bird_fly(bird)     # prints "Started to fly"
make_bird_fly(chicken)  # prints "A chicken can't fly"
make_bird_fly(penguin)  # prints "A penguin can't fly"
```

```
Started to fly
A chicken can't fly
A Penguin can't fly
```

Name mangling with method overriding

Due to name mangling, there is limited support for a valid use-case for class-private members basically to avoid name clashes of names with names defined by subclasses.

Any identifier of the form __geek (at least two leading underscores or at most one trailing underscore) is replaced with _classname__geek, where classname is the current class name with leading underscore(s) stripped.

As long as it occurs within the definition of the class, this mangling is done.

This is helpful for letting subclasses override methods without breaking intraclass method calls.

```
class Map:
        def __init__(self):
                self.__geek()

        def geek(self):
                print("In parent class")

        # private copy of original geek() method
        __geek = geek

class MapSubclass(Map):

        # provides new signature for geek() but
        # does not break __init__()
        def geek(self):
                print("In Child class")

# Driver's code
obj = MapSubclass()
obj.geek()
```

Polymorphism with class methods

- The below code shows how Python can use two different class types, in the same way.
- We create a for loop that iterates through a tuple of objects. Then call the methods without being concerned about which class type each object is.
- We assume that these methods actually exist in each class.

Polymorphism Example

```
class India():
    def capital(self):
        print("New Delhi is the capital of India.")

    def language(self):
        print("Hindi is the most widely spoken language of India.")

    def type(self):
        print("India is a developing country.")

class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")

    def language(self):
        print("English is the primary language of USA.")

    def type(self):
        print("USA is a developed country.")

obj_ind = India()
obj_usa = USA()
for country in (obj_ind, obj_usa):
    country.capital()
    country.language()
    country.type()
```

```
New Delhi is the capital of India.
Hindi is the most widely spoken language of India.
India is a developing country.
Washington, D.C. is the capital of USA.
English is the primary language of USA.
USA is a developed country.
```

Polymorphism with Inheritance

- In Python, Polymorphism lets us define methods in the child class that have the same name as the methods in the parent class.
- In inheritance, the child class inherits the methods from the parent class. However, it is possible to modify a method in a child class that it has inherited from the parent class. This is particularly useful in cases where the method inherited from the parent class doesn’t quite fit the child class.
- In such cases, we re-implement the method in the child class. This process of re-implementing a method in the child class is known as Method Overriding.

Polymorphism in inheritance example

```
class Bird:
  def intro(self):
    print("There are many types of birds.")

  def flight(self):
    print("Most of the birds can fly but some cannot.")

class sparrow(Bird):
  def flight(self):
    print("Sparrows can fly.")

class ostrich(Bird):
  def flight(self):
    print("Ostriches cannot fly.")

obj_bird = Bird()
obj_spr = sparrow()
obj_ost = ostrich()

obj_bird.intro()
obj_bird.flight()

obj_spr.intro()
obj_spr.flight()

obj_ost.intro()
obj_ost.flight()
```

```
There are many types of birds.
Most of the birds can fly but some cannot.
There are many types of birds.
Sparrows can fly.
There are many types of birds.
Ostriches cannot fly.
```

Polymorphism with a Function and objects

- It is also possible to create a function that can take any object, allowing for polymorphism.
- In this example, let's create a function called "func()" which will take an object which we will name "obj". Though we are using the name 'obj', any instantiated object will be able to be called into this function.
- Next, let's give the function something to do that uses the 'obj' object we passed to it. In this case, let's call the three methods, viz., capital(), language() and type(), each of which is defined in the two classes 'India' and 'USA'. Next, let's create instantiations of both the 'India' and 'USA' classes if we don't have them already. With those, we can call their action using the same func() function:

Polymorphism with a Function and objects example

```
class India():
    def capital(self):
        print("New Delhi is the capital of India.")

    def language(self):
        print("Hindi is the most widely spoken language of India.")

    def type(self):
        print("India is a developing country.")

class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")

    def language(self):
        print("English is the primary language of USA.")

    def type(self):
        print("USA is a developed country.")

def func(obj):
    obj.capital()
    obj.language()
    obj.type()

obj_ind = India()
obj_usa = USA()

func(obj_ind)
func(obj_usa)
```

```
def func(obj):
    obj.capital()
    obj.language()
    obj.type()

obj_ind = India()
obj_usa = USA()

func(obj_ind)
func(obj_usa)
```

```
New Delhi is the capital of India.
Hindi is the most widely spoken language of India.
India is a developing country.
Washington, D.C. is the capital of USA.
English is the primary language of USA.
USA is a developed country.
```

Abstraction

Abstract Class

- Abstract classes provide a common interface and behavior for a group of related classes, while leaving the details of their implementation to the subclasses.
- Abstract classes cannot be instantiated and are meant to be subclassed by other classes that provide concrete implementations of their abstract methods.
- Abstract classes define one or more abstract methods, which are methods that have no implementation and must be overridden by any concrete subclass.
- Subclasses that inherit from an abstract class must provide their own implementation of all abstract methods defined in the abstract class, or they will also be considered abstract and cannot be instantiated.
- Abstract classes set strict guidelines or requirements for its subclasses to follow, which can help to ensure that the subclasses implement certain methods or behaviors consistently across different classes.
- Abstract classes can be used to encapsulate the common behavior and interface of a group of related classes, making the code more modular and easier to maintain.
- Abstract classes can also be used to achieve polymorphism, where code can work with objects of different classes without knowing their specific implementations.
- To define an abstract class in Python, you can use the abc module and the ABC class, and decorate abstract methods with the @abstractmethod decorator.

Abstract Mixin with Polymorphism

In this example, the SoundMixin class defines a method make_sound(), which is then implemented differently by each class (Dog, Cat, and Bird).

Each class provides its own implementation of make_sound(), allowing them to make different sounds when the method is called.

```
# Define a mixin class
class SoundMixin:
    def make_sound(self):
        raise NotImplementedError("make_sound() method must be implemented")

# Define classes that use the mixin
class Dog(SoundMixin):
    def make_sound(self):
        return "Woof!"

class Cat(SoundMixin):
    def make_sound(self):
        return "Meow!"

class Bird(SoundMixin):
    def make_sound(self):
        return "Chirp!"

# Example usage
dog = Dog()
print(dog.make_sound())  # Output: Woof!

cat = Cat()
print(cat.make_sound())  # Output: Meow!

bird = Bird()
print(bird.make_sound())  # Output: Chirp!
```

Abstract Mixin with Polymorphism + Encapsulation

The PrintableMixin provides a method printable_info() that generates a printable representation of an object by iterating over its attributes and formatting them as strings.

Shape is an abstract base class (ABC) representing different shapes. It defines two methods: area(), which is expected to be implemented by subclasses to calculate the area of a shape, and print_info(), which is also expected to be implemented but is left abstract.

Circle and Rectangle are specific shape classes that inherit from both the Shape class and the PrintableMixin class. They implement their own __init__ methods to initialize their attributes (radius for Circle and width and height for Rectangle). They also implement the area() method to calculate the area of the shape and the print_info() method to print information about the shape using the mixin's printable_info() method.

```
# Define a mixin class for printable objects
class PrintableMixin:
    def printable_info(self):
        properties = []
        for key, value in self.__dict__.items():
            properties.append(f"{key}: {value}")
        return f"{self.__class__.__name__}({', '.join(properties)})"

# Define an abstract base class representing different shapes
class Shape:
    def area(self):
        raise NotImplementedError("area() method must be implemented")

    def print_info(self):
        raise NotImplementedError("print_info() method must be implemented")

# Define specific shape classes
class Circle(Shape, PrintableMixin):
    def __init__(self, radius):
        self.__radius = radius

    def get_radius(self):
        return self.__radius

    def area(self):
        return 3.14 * self.__radius ** 2

    def print_info(self):
        return self.printable_info()

class Rectangle(Shape, PrintableMixin):
    def __init__(self, width, height):
        self.__width = width
        self.__height = height

    def get_width(self):
        return self.__width

    def get_height(self):
        return self.__height

    def area(self):
        return self.__width * self.__height

    def print_info(self):
        return self.printable_info()

# Example usage
circle = Circle(5)
print(circle.print_info())  # Output: Circle(radius: 5)
print("Radius:", circle.get_radius())  # Output: Radius: 5

rectangle = Rectangle(3, 4)
print(rectangle.print_info())  # Output: Rectangle(width: 3, height: 4)
print("Width:", rectangle.get_width())  # Output: Width: 3
print("Height:", rectangle.get_height())  # Output: Height: 4
```

Additional topics that may be read

- Decorators and Property
- Abstract Base Class