OOPS - Encapsulation

August 24, 2022

Encapsulation In Python

What Is Meant By Encpsulation ?

Encapsulation in Python is the process of wrapping up variables and methods into a single entity

In programming, a class is an example that wraps all the variables and methods defined inside it

By doing so, you can hide the internal state of the object from the outside. This is known as Information Hiding.

How To Use Encapsulation In Python :

Encapsulation means restricting the access of methods and variables

We add the direct access and change restriction feature to methods or variables

Why Are We Doing This :

The codes we write should not be changed or the values we change should be changed in a controlled manner.

Benefits of Encapsulation :

Encapsulation provides well-defined, readable code.

Prevents Accidental Modification or Deletion.

Encapsulation provides security.

It also ensures that objects are self-sufficient functioning pieces and can work independently.

[3]:
```
#The following defines the Counter class:
class Counter:
    #constructer
    def __init__(self):
        self.current = 0

    def increment(self):
        self.current += 1

    def value(self):
        return self.current

    def reset(self):
```

```
        self.current = 0
```

[4]:
```
#The following creates a new instance of the Counter class and calls the␣
 ↪increment():
counter = Counter()

counter.increment()
counter.increment()
counter.increment()

print(counter.value())
```

```
3
```

[6]:
```
#From the outside of the Counter class,
# We still can access the current attribute and change it to whatever you wan
counter = Counter()

counter.increment()
counter.increment()
counter.current = -999

print(counter.value())
```

```
-999
```

Access Modifiers in Python :

Encapsulation can be achieved by declaring the data members and methods of a class either as private or protected

But In Python, we don't have direct access modifiers like public, private, and protected.

We can achieve this by using single underscore and double underscores.

Access modifiers limit access to the variables and methods of a class.

Python provides three types of access modifiers private, public, and protected :

Public Member : Accessible anywhere from outside class.

Private Member : Accessible within the class.

Protected Member : Accessible within the class and its sub-classes.

Public Member :

Public data members are accessible within and outside of a class.

All member variables of the class are by default public.

[8]:
```
class Employee:
    # constructor
    def __init__(self, name, salary):
```

```
        # public data members
        self.name = name
        self.salary = salary

    # public instance methods
    def show(self):
        # accessing public data member
        print("Name: ", self.name, 'Salary:', self.salary)

# creating object of a class
emp = Employee('Nitheesh', 30000)

# accessing public data members
print("Name: ", emp.name, 'Salary:', emp.salary)

# calling public method of the class
emp.show()
```

```
Name:  Nitheesh Salary: 30000
Name:  Nitheesh Salary: 30000
```

Private Member :

We can protect variables in the class by marking them private.

To define a private variable add two underscores as a prefix at the start of a variable name.

Private members are accessible only within the class, and we can't access them directly from the class objects.

[1]:
```
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data member
        self.name = name
        # private member
        self.__salary = salary

# creating object of a class
emp = Employee('Nitheesh', 30000)

# accessing private data members
print('Salary:', emp.__salary)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-1-50f8e8ff4fab> in <module>
     11
     12 # accessing private data members
---> 13 print('Salary:', emp.__salary)
```

```
AttributeError: 'Employee' object has no attribute '__salary'
```

Access Private Members

Create public method to access private members

Use name mangling

Public method to access private members

[5]:
```
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data member
        self.name = name
        # private member
        self.__salary = salary

    # public instance methods
    def show(self):
        # private members are accessible from a class
        print("Name: ", self.name, 'Salary:', self.__salary)

# creating object of a class
emp = Employee('Nitheesh', 30000)

# calling public method of the class
emp.show()
```

```
Name:  Nitheesh Salary: 30000
```

Name Mangling to access private members

The name mangling is created on an identifier by adding two leading underscores and one trailing underscore.

_classname___dataMember, where classname is the current class, and data member is the private variable name.

[6]:
```
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data member
        self.name = name
        # private member
        self.__salary = salary

# creating object of a class
emp = Employee('Nitheesh', 30000)
```

```
print('Name:', emp.name)
# direct access to private member using name mangling
print('Salary:', emp._Employee__salary)
```

```
Name: Nitheesh
Salary: 30000
```

Protected Member :

Protected members are accessible within the class and also available to its sub-classes.

To define a protected member, prefix the member name with a single underscore _.

These are used when we implement inheritance and want data members access to only child classes.

[7]:
```
# base class
class Company:
    def __init__(self):
        # Protected member
        self._project = "NLP"

# child class
class Employee(Company):
    def __init__(self, name):
        self.name = name
        Company.__init__(self)
    #public method
    def show(self):
        print("Employee name :", self.name)
        # Accessing protected member in child class
        print("Working on project :", self._project)

c = Employee("Nitheesh")
c.show()

# Direct access protected data member
print('Project:', c._project)
```

```
Employee name : Nitheesh
Working on project : NLP
Project: NLP
```

Getters and Setters in Python

To implement proper encapsulation in Python, we need to use setters and getters

Use the getter method to access data members and the setter methods to modify the data members.

In Python, private variables are not hidden fields like in other programming languages

The getters and setters methods are often used when :

When we want to avoid direct access to private variables

To add validation logic for setting a value

[13]:
```python
class Student:
    def __init__(self, name, age):
        self.name = name
        # private member
        self.__age = age

    # getter method
    def get_age(self):
        return self.__age

    # setter method
    def set_age(self, age):
        self.__age = age

stud = Student('Nitheesh', 25)

# retrieving age using getter
print('Name:', stud.name, stud.get_age())

# changing age using setter
stud.set_age(21)

# retrieving age using getter
print('Name:', stud.name, stud.get_age())
```

```
Name: Nitheesh 25
Name: Nitheesh 21
```

[8]:
```python
#Information Hiding and conditional logic for setting an object attributes
class Student:
    def __init__(self, name, roll_no, age):
        # public member
        self.name = name
        # private members to restrict access
        # avoid direct data modification
        self.__roll_no = roll_no
        self.__age = age

    def show(self):
        print('Student Details:', self.name, self.__roll_no)

    # getter methods
    def get_roll_no(self):
        return self.__roll_no
```

```
    # setter method to modify data member
    # condition to allow data modification with rules
    def set_roll_no(self, number):
        if number > 50:
            print('Invalid roll no. Please set correct roll number')
        else:
            self.__roll_no = number

Nit = Student('Nitheesh', 10, 15)

# before Modify
Nit.show()
# changing roll number using setter
Nit.set_roll_no(120)


Nit.set_roll_no(25)
Nit.get_roll_no()
```

```
Student Details: Nitheesh 10
Invalid roll no. Please set correct roll number
```

[8]: 25

Using @property decorators to achieve getters and setters behaviour

In Python property( )is a built-in function that creates and returns a property object.

A property object has three methods, getter(), setter(), and delete().

[11]:
```
# Python program showing the use of @property

class Student:
    def __init__(self):
        self.__age = 0
    # using property decorator a getter function
    @property
    def age(self):
        print("getter method called")
        return self.__age

    # a setter function
    @age.setter
    def age(self, a):
        if(a < 18):
            raise ValueError("Sorry you age is below eligibility criteria")
        print("setter method called")
        self.__age = a
```

```
mark = Student()
mark.age = 19

print(mark.age)
```

```
setter method called
getter method called
19
```

Encapsulation In Python FAQs :

What is Encapsulation in Python ?

How can we achieve Encapsulation in Python ?

How can we define a variable as Private ?

How can we define a variable as Protected ?

What are acess modifiers ?