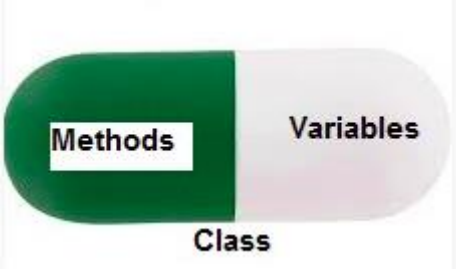
ENCAPSULATION

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.



In the real world, consider your college as an example. Each and every department's student records are restricted only within the respective department. CSE department faculty cannot access the ECE department student record and so on. Here, the department acts as the class and student records act like variables and methods.

Why Do We Need Encapsulation?

Encapsulation acts as a protective layer by ensuring that, access to wrapped data is not possible by any code defined outside the class in which the wrapped data are defined. Encapsulation provides security by hiding the data from the outside world.

In Python, Encapsulation can be achieved by declaring the data members of a class either as private or protected. In Python, 'Private' and 'Protected' are called Access Modifiers, as they modify the access of variables or methods defined in a class.

Encapsulation Using Private Members

If we declare any variable or method as private, then they can be accessed only within the class in which they are defined. In Python, private members are preceded by two underscores.

Consider the example given below:

```
class A:
   __x = 10 #private variable

   def __init__(self):
      print(self.__x) #It will print as we are accessing x inside class only

  def __m1(self):
     print('m1—A')

a = A()
```

```
print(a.__x) # AttributeError: 'A' object has no attribute '__x'
a.__m1()  # AttributeError: 'A' object has no attribute '__m1'
```

As from the above example, we can see that private variable and method cannot be accessed outside the class. So, to access them outside class there are 2 ways, by **Name mangling and by Setter-Getter.**

Accessing through setter-getter:

Setter is used to initialize the instance variable and Getter is used to get that particular instance variable.

```
class Student:
    def setRollno(self,rollno):
        self.__rollno = rollno

    def getRollno(self):
        return self.__rollno

    def setName(self,name):
        self.__name = name

    def getName(self):
        return self.__name

s = Student()
s.setRollno(int(input('Enter rollno : ')))
s.setName(input('Enter name : '))

print('Rollno : ',s.getRollno())
print('Name : ',s.getName())
```

OUTPUT:

```
Enter rollno : 1
Enter name : abc
Rollno :  1
Name :  abc
```

Accessing through Name mangling:

Syntax:
objName._className__varName/methodName

```
class A:
    __x = 10 #private variable
    def __init__(self):
        print(self.__x)
```

```
    def __m1(self):
        print('m1--A')

a = A()
print(a._A__x)   #Name mangling
a._A__m1()   #Name mangling
```

OUTPUT:

```
10
10
m1—A
```

Private members in Inheritance:

In inheritance, private members will be accessed only within that class where they are coded. For e.g.,

```
class A:
    def __m1(self):
        print('m1--A')

    def m2(self):
        print('m2--A')
        self.__m1() #self=b, Here it will print m1--A as private method is accessible in that class
                    #A only. It will not go in class B even if self contains class B object

class B(A):
    def __m1(self):
        print('m1--B')

b = B()
#b.__m1() Error
b.m2()
```

OUTPUT:

```
m2--A
m1—A
```

Encapsulation Using Protected Members

Protected members can be accessed within the class in which they are defined and also within the derived classes. In Python, protected members are preceded by a single underscore. Using a single leading underscore is merely a convention.

As mentioned, it is just a convention and a leading underscore doesn't actually make any variable or method protected. It's just that if you see a variable or method with a leading underscore in Python code you should follow the convention that it should be used internally within a class.

Consider the example given below:

```
class A:
    _x = 10 #protected variable

    def __init__(self):
        print(self._x)

    def _m1(self):
        print('m1--A')

a = A()
print(a._x)
a._m1()
```

OUTPUT:

```
10
10
m1—A
```

Protected members in Inheritance:

In inheritance, protected members will be accessed in that class where they are coded and also in its child. For e.g.,

```
class A:
    def _m1(self):
        print('m1--A')

    def m2(self):
        print('m2--A')
        self._m1() #self=b, so control will go in class B's m1 method

class B(A):
    def _m1(self):
        print('m1--B')
```

```
        super()._m1()

b = B()
b._m1()
print('--------------------')
b.m2()
```

OUTPUT:

```
m1--B
m1--A
--------------------
m2--A
m1--B
m1--A
```