ANNEXURE

7

Getters, Setters, @property, and @deleter

The `getters` and `setters` methods are used in many object oriented programming languages to provide data encapsulation (binding data and functions in a single entity). They are also known as ***mutator*** methods. While `getter` method is used for retrieving data, `setter` method, on the other hand, is used to set a new value for the data (for changing the data). In OOP languages, the attributes of a class are made private to hide and protect them from other code. But Python usually has all attributes as public (except those which starts with a double underscore).

Example A7.1 Program that uses `get` and `set` functions to retrieve and set a value

```
class Sample:
def __init__(self,val):
      self.val = val
def get_val(self):
      return self.val
def set_val(self,val):
      self.val = val
S = Sample(20)
S.set_val(10)
print(S.get_val())
```

OUTPUT

```
10
```

However, there is no data encapsulation in the above code. Data can be openly accessed from anywhere in the program. In C++ and Java, private attributes are used with `getters` and `setters`. But to support encapsulation in the true sense, Python supports a class with a `@property` and `@setter` decorators. The program given below demonstrates the use of these decorators.

Example A7.2 Program to demonstrate `@property` and `@setter`

```
class Sample:
def __init__(self,val):
      self.val = val
   @property
def val(self):
      return self.__val
   @val.setter
```

```
def val(self,val):
      self.__val = val
S = Sample(20)
S.val = 100
print(S.val)
```

OUTPUT

```
100
```

In the aforementioned program, note that a method which is used for getting a value is decorated with `@property`. Similarly, the method which sets a value of an instance variable—`setter` is decorated with `@x.setter`, where `x` is the name of the function.

However, there is another way of doing the same task but without using decorators to define the property. Such a code is given below.

Example A7.3 Program for getting and setting value by using `getter` and `setter` methods

```
class Sample:
def __init__(self,val):
      self.val = val
def get_val(self):
      return self.__val
def set_val(self,val):
      self.__val = val
   val = property(get_val, set_val)
S = Sample(20)
S.val = 100
print(S.val)
```

OUTPUT

```
100
```

To provide better data hiding, we can even make our `getter` and `setter` methods as *private*. This is illustrated in the code given below.

Example A7.4 Program to demonstrate private `getter` and `setter` methods

```
class Sample:
def __init__(self,val):
      self.__val = val
def __get_val(self):
      return self.__val
def __set_val(self,val):
      self.__val = val
   val = property(__get_val, __set_val)
```

```
S = Sample(20)
S.val = 100
print(S.val)
```

OUTPUT

```
100
```

From the previous discussion, we have seen that each attribute has or should have its own property (or `getter-setter` pair). Let us take another example. Suppose we created a class `Student` with attributes—`first_name` and `last_name`, but at a later point of time we want to change the class to have a combined name. For this we can write a method called `name` which returns the full name using `@property` decorator which lets a method behave like an attribute. The code illustrating this concept is given below.

Example A7.5 Program to demonstrate a method behaving like an attribute

```
class Student:
def __init__(self, first_name, last_name):
      self.__first_name = first_name
      self.__last_name = last_name
   @property
def name(self):
      return "%s %s" % (self.__first_name, self.__last_name)
S = Student("Abdul", "Kalam")
print(S.name)
```

OUTPUT

```
Abdul Kalam
```

Finally, we have also have a decorator `deleter` for our attribute. The key role of a `deleter` is to delete the attribute from our object. However, note that the `getter`, `setter`, and `deleter` methods must all have the same name as shown in the program given below.

Example A7.6 Program to demonstrate `deleter` method

```
class Sample:
def __init__(self,val):
      self.val = val
   @property
def val(self):
      return self.__val
   @val.setter
def val(self,val):
      self.__val = val
   @val.deleter
def val(self):
```

```
    del self.val
S = Sample(20)
S.val = 100
print(S.val)
```

OUTPUT

```
100
```

Before concluding the topic, let us write another program.

Example A7.7 Program that implements a class `Square` having a `getter` function to return the area of a square

```
class Square:
def __init__(self, side):
    self.__side = side
  @property
def area(self):    # area method acts as getter
    return self.__side * self.__side
S = Square(9)
print(S.area)  # area is a method which works like an attribute
```

OUTPUT

```
81
```