Python Properties
Getter, setter and deleter

Introduction

It is often considered best practice to create getters and setters for a class's public properties. Many languages allow you to implement this in different ways, either by using a function (like person.getName()), or by using a language-specific get or set construct. In Python, it is done using @property.

Getters and setters are used in many object oriented programming languages to ensure the principle of data encapsulation. They are known as mutator methods as well.

The syntax of property() method is:

```
property(fget=None, fset=None, fdel=None, doc=None)
```

property() Parameters

The property() method takes four optional parameters:

- **fget (Optional)** - function for getting the attribute value
- **fset (Optional)** - function for setting the attribute value
- **fdel (Optional)** - function for deleting the attribute value
- **doc (Optional)** - string that contains the documentation (docstring) for the attribute

Return value from property()

The property() method returns a property attribute from the given getter, setter and deleter.

- If no arguments are given, property() method returns a base property attribute that doesn't contain any getter, setter or deleter.
- If **doc** isn't provided, property() method takes the docstring of the getter function.

First, we demonstrate in the following example, how we can design a class in a Javaesque way with getters and setters to encapsulate the private attribute "self.__x":

```
class P:

    def __init__(self,x):
        self.__x = x

    def get_x(self):
        return self.__x

    def set_x(self, x):
        self.__x = x
```

We can see in the following demo session how to work with this class and the methods:

```
>>> from mutators import P
>>> p1 = P(42)
>>> p2 = P(4711)
>>> p1.get_x()
42
>>> p1.set_x(47)
>>> p1.set_x(p1.get_x()+p2.get_x())
>>> p1.get_x()
4758
>>>
```

But if we want to restrict the value of x according to our needs?

It is easy to change our first P class to cover this problem. We change the set_x method accordingly:

```
class P:

    def __init__(self,x):
        self.set_x(x)

    def get_x(self):
        return self.__x

    def set_x(self, x):
        if x < 0:
            self.__x = 0
        elif x > 1000:
            self.__x = 1000
        else:
            self.__x = x
```

The following Python session shows that it works the way we want it to work:

```
>>> from mutators import P
>>> p1 = P(1001)
>>> p1.get_x()
1000
>>> p2 = P(15)
>>> p2.get_x()
15
>>> p3 = P(-1)
>>> p3.get_x()
0
```

But there is a catch: Let's assume we have designed our class with the public attribute and no methods. People have already used it a lot and they have written code like this:

```
from p import P
p1 = P(42)
p1.x = 1001
```

Our new class means breaking the interface. The attribute x is not available anymore. That's why in Java e.g. people are recommended to use only private attributes with getters and setters, so that they can change the implementation without having to change the interface.

What do you think about the expression "p1.set_x(p1.get_x()+p2.get_x())"? It's ugly, isn't it? It's a lot easier to write an expression like the following, if we had a public attribute x:

```
p1.x = p1.x + p2.x
```

But Python offers a solution to this problem. The solution is called properties!

The class with a property looks like this:

```
class P:

    def __init__(self,x):
        self.x = x

    @property
    def x(self):
        return self.__x

    @x.setter
    def x(self, x):
        if x < 0:
            self.__x = 0
        elif x > 1000:
            self.__x = 1000
        else:
            self.__x = x
```

A method which is used for getting a value is decorated with "@property", i.e. we put this line directly in front of the header. The method which has to function as the setter is decorated with "@x.setter". If the function had been called "f", we would have to decorate it with "@f.setter".

Two things are noteworthy: We just put the code line "self.x = x" in the __init__ method and the property method x is used to check the limits of the values. The second interesting thing is that we wrote "two" methods with the same name and a different number of parameters "def x(self)" and "def x(self,x)". It works here due to the decorating:

```
>>> from p import P
>>> p1 = P(1001)
>>> p1.x
1000
>>> p1.x = -12
>>> p1.x
0
>>>
```

Alternatively, we could have used a different syntax without decorators to define the property. As you can see, the code is definitely less elegant and we have to make sure that we use the getter function in the __init__ method again:

```
class P:

    def __init__(self,x):
        self.set_x(x)

    def get_x(self):
        return self.__x

    def set_x(self, x):
        if x < 0:
            self.__x = 0
        elif x > 1000:
            self.__x = 1000
        else:
            self.__x = x

    x = property(get_x, set_x)
```

Example 1: Create attribute with getter, setter and deleter using property()

```
class Person:
      def __init__(self, name):
          self._name = name

      def getName(self):
          print('Getting name')
          return self._name

      def setName(self, value):
          print('Setting name to ' + value)
          self._name = value

          def delName(self):
          print('Deleting name')
          del self._name

      # Set property to use getName, setName
      # and delName methods
      name = property(getName, setName, delName, 'Name property')


p = Person('Adam')
print(p.name)

p.name = 'John'

del p.name
```

When you run the program, the output will be:

```
Getting name
```

```
The name is: Adam
Setting name to John
Deleting name
```

Here, _name is used as the private variable for storing the name of a Person.
We also set:

- a getter method getName() to get the name of the person,
- a setter method setName() to set the name of the person,
- a deleter method delName() to delete the name of the person.

Now, we set a new property attribute name by calling the property() method.
As shown in the program, referencing p.name internally calls getName() as getter, setName() as setter and delName() as deleter through the printed output present inside the methods.

Example 2: Create attribute with getter, setter and deleter using @property decorator

Instead of using the property() method, you can use the Python decorator @property to assign the getter, setter and deleter.

```
class Person:
    def __init__(self, name):
        self._name = name

    @property
    def name(self):
        print('Getting name')
        return self._name

    @name.setter
    def name(self, value):
        print('Setting name to ' + value)
        self._name = value

    @name.deleter
    def name(self):
        print('Deleting name')
        del self._name

p = Person('Adam')
print('The name is:', p.name)

p.name = 'John'

del p.name
```

Here, instead of using the property() method, we've used the @property decorator.
First, we specify that name() method is also an attribute of Person. This is done by using @property before the getter method as shown in the program.
Next, we use the attribute name to specify the setter and the deleter. This is done by using @<name-of-attribute>.setter (@name.setter) for setter method and @<name-of-attribute>.deleter (@name.deleter) for deleter method.
Notice, we've used the same method name() with different definitions for defining the getter, setter and deleter.
Now, whenever we use p.name, it internally calls the appropriate getter, setter and deleter as shown by the printed output present inside the method.