Chapter 23
Python Properties

23.1 Introduction

Many object-oriented languages have the explicit concept of encapsulation; that is the ability to hide data within an object and only to provide specific gateways into that data. These gateways are methods defined to *get* or *set* the value of an attribute (often referred to as getters and setters). This allows more control over access to the data; for example, it is possible to check that only a positive integer above zero, but below 120, is used for a person's age etc.

In many languages such as Java and C# attributes can be hidden from external access using specific keywords (such as private) that indicate the data should be made private to the object.

Python does not explicitly have the concept of encapsulation; instead it relies on two things; a standard convention used to indicate that an attribute should be considered private and a concept called a property which allows setters and getters to be defined for an attribute.

23.2 Python Attributes

All object attributes are publicly available in Python; that is, they are all visible to any code using the object.

For example, given the following definition of the class `Person` both `name` and `age` are part of the public interface of the class `Person`;

J. Hunt, *A Beginners Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-20290-3_23

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

    def __str__(self):
        return 'Person[' + str(self.name) + '] is ' +
                           str(self.age)
```

Because `name` and `age` are part of the class's public interface it means that we can write:

```
person = Person('John', 54)
person.name = 42
person.age = -1
print(person)
```

Which is of course a bit bizarre as the person now has the name '42' and an age of −1, thus the output from this is:

```
Person[42] is -1
```

We can indicate that we want to treat age and name as being private to the object by prefixing the attribute names with an underbar ('_') as shown below:

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

    def __str__(self):
        return 'Person[' + str(self._name) +'] is ' + s
                             str(self._age)
```

This tells Python programmers that we want to consider `_name` and `_age` as being *private*. However, it should be noted that this is only a *convention*; albeit a *very strongly* adhered to convention. There is still nothing here that would stop someone writing:

```
person = Person('John', 54)
person._age = -1
print(person)
```

However, the developer of the class `Person` is at liberty to change the internals of the class (such as `_age`) without notice and most would consider that anyone who had ignored the convention and now had a problem had only themselves to blame.

23.3 Setter and Getter Style Methods

This of course raises the question; how should we now get hold of a Persons' `name` and `age` in an acceptable way?

The answer is that a developer should provide *getter* methods and *setter* methods that can be used to access the values.

We can update the `Person` class with some getter methods and a single setter method:

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

    def get_age(self):
        return self._age

    def set_age(self, new_age):
        if isinstance(new_age,int) & new_age > 0 &
new_age < 120:
            self._age = new_age

    def get_name(self):
        return self._name

    def __str__(self):
        return 'Person[' + str(self._name) +'] is ' +
str(self._age)
```

The two getter methods have the format of `get_` followed by the name of the attribute they are getting. Thus, we have `get_age` and `get_name`. Typically, all that getters do is to return the attribute being used (as is the case here).

The single setter method is a little different; it validates the data that has been provided to check that it is appropriate (i.e. that it is an Integer using `isinstance (new_age,int)` and that it is a value over Zero but under 120). Only if the data passes these checks is it used as the new value of the person's age, for example if we try to set a persons age to −1:

```
person = Person('John', 54)
person.set_age(-1)
print(person)
```

Then this is ignored, and the person's age remains as it was, thus the output of this is:

```
Person[John] is 54
```

It should be noted that this might be considered *silent failure*; that is we tried to set the `age` and it failed but no one knows this. In many cases rather than fail silently we would prefer to notify someone of the error by throwing some form of Error object; this will be discussed in the next chapter on Error and Exception handling.

You might well ask at this point where is the setter for the `_name` attribute? The answer is that we want to make the `_name` attribute a *read only* attribute and therefore we have not provided a setter style method. This is a common idiom followed in Python—you can have read-write attributes and read-only attributes depending on whether they have getter and setter methods or not. It is also possible to a *write-only* attribute, but this is very rare and only has a few use cases.

23.4 Public Interface to Properties

Although we now have a more formal interface to the attributes held by an instance of the class `Person`; it is rather ungainly:

```
person = Person('John', 54)
print(person)
print(person.get_age())
print(person.get_name())
```

We end up having to write more code and although there is an argument that it makes the code more obvious (i.e. `person.get_age()` can be read as get the age of the person object); it is somewhat verbose and you have to remember to include the parentheses (`()`').

To get around this a concept known as Properties was introduced in Python 2.2. In the original syntax for this it was possible to add an addition line of code to the class that told Python that you wanted to provide a new property and that specific methods were to be used to set and get the values of this property.

The syntax for defining a property in this way is:

```
<property_name> = property(fget=None, fset=None,
 fdel=None, doc=None)
```

Where `fget` indicates the getter function, `fset` the setter function `fdel` the function to be used to delete a value and `doc` provides documentation on the property (all of which are optional).

We can modify our `Person` class so that `age` is now a property (note a common convention is that if the attribute is named `_age`, the methods are named `get_age` and `set_age` and the property will be called `age`):

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

   def get_age(self):
       return self._age
   def set_age(self, new_age):
       if isinstance(new_age,int) & new_age > 0 &
new_age < 120:
           self._age = new_age

   age = property(get_age, set_age, doc="An age property")

   def get_name(self):
       return self._name

   name = property(get_name, doc="A name property")

   def __str__(self):
       return 'Person[' + str(self._name) +'] is ' +
                          str(self._age)
```

We can now write:

```
person = Person('John', 54)
print(person)
print(person.age)
print(person.name)
person.age = 21
print(person)
```

Notice how we can now write `person.age` and `person.age = 21`; in both these cases we are accessing the *property* age that results in the method `get_age()` and `set_age()` being executed respectively. Thus, the setter is still protecting the update to the underlying `_age` attribute that is actually used to hold the actual value.

Also note that if a method is not provided for one of the `fget`, `fset`, `fdel` methods then this is not an error; it merely indicates that the property does not support that type of accessor. Thus, the `name` property is a *read_only* property as it does not define a setter method.

A delete method can be used to release the memory associated with an attribute; in the case of an `int` it is not required but it may be required for a more complex, user defined type.

We could therefore write:

```
def del_name(self):
        del self._name
name = property(get_name, fdel=del_name, doc="A name
property")
```

Note that we are using a keyword reference for the delete method as we have skipped the setter and cannot therefore rely on positional arguments.

23.5 More Concise Property Definitions

The example shown in the previous section works but it is still quite verbose itself; while this is on the class writers' side it still seems somewhat heavyweight.

To overcome this a more concise option has been available since Python 2.4. This approach uses what are known as decorators. Decorators represent meta data (that is information about your code that the Python interpreter can use to work out what you want it to do with certain things).

Python 2.4 introduced three new decorators `@property`, `@<property-name>.setter` and `@<propertyname>.deleter`. These decorators are added to the start of a method definition to indicate that the method should be used to provide access to a property (and define that property), define a setter for the property or a deleter for the property.

We will now update our `Person` class to use the decorators:

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

    @property
    def age(self):
        """ The docstring for the age property """
        print('In age method')
        return self._age

    @age.setter
    def age(self, value):
        print('In set_age method')
        if isinstance(value,int) & value > 0 & value < 120:
            self._age = value

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

    @name.deleter
    def name(self):
        del self._name

    def __str__(self):
        return 'Person[' + str(self._name) +'] is ' +
str(self._age)
```

Notice three important things about this example:

- The name of the methods is no longer a `set_age` and `get_age`; instead both methods are now just `age` and the decorator distinguishes their role. Also notice that we no longer have a separate statement that declares the property—it is now implicit in the use of the `@property` decorator and the name of the associated method.
- The `@property` decorator is used to define the name of the property (in this case age) and to define further decorators which will be named after the property with a `.setter` or `.deleter` element e.g. `@age.setter`.
- The documentation string is now defined in the method associated with the `@property` decorator (providing this documentation string is usually considered good practice).

However, we do not need to change the program that used the class `Person`, as the interface to the class remained the same.

23.6 Online Resources

Some online resources on properties are:

- https://www.python-course.eu/python3_properties.php a discussion on Python properties versus getters and setters.
- https://www.journaldev.com/14893/python-property-decorator a short introduction to the `@property` decorator.

23.7 Exercises

In this exercise you will add *properties* to an existing class.

Return to the `Account` class that you created several chapters ago; convert the `balance` into a read only property using decorators, then verify that the following program functions correctly:

```
acc1 = CurrentAccount('123', 'John', 10.05, 100.0)
acc2 = DepositAccount('345', 'John', 23.55, 0.5)
acc3 = acc3 = InvestmentAccount('567', 'Phoebe', 12.45,
'high risk')

print(acc1)
print(acc2)
print(acc3)

acc1.deposit(23.45)
acc1.withdraw(12.33)
print('balance:', acc1.balance)

print('Number of Account instances created:',
Account.instance_count)

print('balance:', acc1.balance)
acc1.withdraw(300.00)
print('balance:', acc1.balance)
```

The output from this might be:

```
Creating new Account
Creating new Account
Creating new Account
Account[123] - John, current account = 10.05overdraft
limit: -100.0
Account[345] - John, savings account = 23.55interest
rate: 0.5
Account[567] - Phoebe, investment account = 12.45
balance: 21.17
Number of Account instances created: 3
balance: 21.17
Withdrawal would exceed your overdraft limit
balance: 21.17
```