Attribute Access

All Versions

Examples

Basic Attribute Access using the Dot Notation

Let's take a sample class.

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

book1 = Book(title="Right Ho, Jeeves", author="P.G. Wodehouse")
```

In Python you can access the attribute *title* of the class using the dot notation.

```
>>> book1.title
'P.G. Wodehouse'
```

If an attribute doesn't exist, Python throws an error:

```
>>> book1.series
Traceback  (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'Book' object has no attribute 'series'
```

Setters, Getters & Properties

For the sake of data encapsulation, sometimes you want to have an attribute which value comes from other attributes or, in general, which value shall be computed at the moment. The standard way to deal with this situation is to create a method, called getter or a setter.

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author
```

In the example above, it's easy to see what happens if we create a new Book that contains a title and a author. If all books we're to add to our Library have authors and titles, then we can skip the getters and setters and use the dot notation. However, suppose we have some books that do not have an author and we want to set the author to "Unknown". Or if they have multiple authors and we plan to return a list of authors.

In this case we can create a getter and a setter for the *author* attribute.

```
class P:
    def __init__(self,title,author):
        self.title = title
        self.setAuthor(author)

    def get_author(self):
        return self.author

    def set_author(self, author):
        if not author:
            self.author = "Unknown"
        else:
            self.author = author
```

This scheme is not recommended.

One reason is that 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:

```
>>> book = Book(title="Ancient Manuscript", author="Some Guy")
>>> book.author = ""  #Cos Some Guy didn't write this one!
```

Now we have a problem. Because *author* is not an attribute! Python offers a solution to this problem called properties. A method to get properties is decorated with the @property before it's header. The method that we want to function as a setter is decorated with @attributeName.setter before it.

Keeping this in mind, we now have our new updated class.

```
class Book:
```

```
    def __init__(self, title, author):
        self.title = title
        self.author = author

    @property
    def author(self):
        return self.__author

    @author.setter
    def author(self, author):
        if not author:
            self.author = "Unknown"
        else:
            self.author = author
```

Note, normally Python doesn't allow you to have multiple methods with the same name and different number of parameters. However, in this case Python allows this because of the decorators used.

If we test the code:

```
>>> book = Book(title="Ancient Manuscript", author="Some Guy")
>>> book.author = ""  #Cos Some Guy didn't write this one!
>>> book.author
Unknown
```

Syntax

```
x.title # Accesses the title attribute using the dot notation
```

```
x.title = "Hello World" # Sets the property of the title attribute using the dot notation
```

```
@property # Used as a decorator before the getter method for properties
```

```
@title.setter # Used as a decorator before the setter method for properties
```

Parameters

Remarks