Introduction to Python descriptors

Manage attribute access with Python descriptors

Alex Starostin June 26, 2012

Learn to easily create and apply descriptors in Python.

Introduction

Python descriptors were introduced in Python 2.2, along with new style classes, yet they remain widely unused. Python descriptors are a way to create managed attributes. Among their many advantages, managed attributes are used to protect an attribute from changes or to automatically update the values of a dependant attribute.

Descriptors increase an understanding of Python, and improve coding skills. This article introduces descriptor protocol and demonstrates how to create and use descriptors.

Descriptors protocol

Python *descriptor protocol* is simply a way to specify what happens when an attribute is referenced on a model. It allows a programmer to easily and efficiently manage attribute access:

- set
- get
- delete

In other programming languages, descriptors are referred to as *setter* and *getter*, where public functions are used to Get and Set a private variable. Python doesn't have a private variables concept, and descriptor protocol can be considered as a Pythonic way to achieve something similar.

In general, a descriptor is an object attribute with a binding behavior, one whose attribute access is overridden by methods in the descriptor protocol. Those methods are `__get__`, `__set__`, and `__delete__`. If any of these methods are defined for an object, it is said to be a descriptor. Take a closer look at these methods in Listing 1.

Listing 1. Descriptor methods

```
__get__(self, instance, owner)
__set__(self, instance, value)
__delete__(self, instance)
```

Where:

`__get__` accesses the attribute. It returns the value of the attribute, or raise the `AttributeError` exception if a requested attribute is not present.

`__set__` is called in an attribute assignment operation. Returns nothing.

`__delete__` controls a delete operation. Returns nothing.

It is important to note that descriptors are assigned to a class, not an instance. Modifying the class overwrites or deletes the descriptor itself, rather than triggering its code.

When descriptors are needed

Consider an `email` attribute. Verification of the correct email format is necessary before assigning a value to that attribute. This descriptor allows email to be processed through a regular expression and its format validated before assigning it to an attribute.

In many other cases, Python protocol descriptors control access to attributes, such as protection of the `name` attribute.

Creating descriptors

You can create a descriptor a number of ways:

- Create a class and override any of the descriptor methods: `__set__`, `__ get__`, and `__delete__`. This method is used when the same descriptor is needed across many different classes and attributes, for example, for type validation.
- Use a property type which is a simpler and more flexible way to create a descriptor.
- Use the power of property decorators which are a combination of property type method and Python decorators.

All examples below are similar from an operational viewpoint. The difference lies in implementation.

Creating descriptors using class methods

Listing 2 demonstrates the simplicity of controlling attribute assignment in Python.

Listing 2. Creating descriptors using class methods

```
class Descriptor(object):

    def __init__(self):
        self._name = ''

    def __get__(self, instance, owner):
        print "Getting: %s" % self._name
        return self._name

    def __set__(self, instance, name):
        print "Setting: %s" % name
        self._name = name.title()

    def __delete__(self, instance):
        print "Deleting: %s" %self._name
        del self._name

class Person(object):
    name = Descriptor()
```

Use this code and see the output:

```
>>> user = Person()
>>> user.name = 'john smith'
Setting: john smith
>>> user.name
Getting: John Smith
'John Smith'
>>> del user.name
Deleting: John Smith
```

A descriptor class was created overriding `__set__()`, `__get__()` and `__delete__()` methods of the parent class in such a way that

- `get` will print *Getting*
- `delete` will print *Deleting*
- `set` will print *Setting*

and change the attribute value to title (first letter uppercase, other letters lowercase) before assignment. This is handy, for example, when storing and printing names.

Uppercase conversion can equally be moved to `__get__()` method. The `_value` will have the original value, and will be converted to title on `get` request.

Creating descriptors using property type

While the descriptor specified in Listing 2 is valid and functional, another method is through the property type. With the property(), it is easy to create a usable descriptor for any attribute. The syntax for creating `property()` is `property(fget=None, fset=None, fdel=None, doc=None)` where:

- fget – attribute get method
- fset – attribute set method
- fdel – attribute delete method

- doc – docstring

Rewrite the example using property, as in Listing 3.

Listing 3. Creating descriptor with property type

```
class Person(object):
    def __init__(self):
        self._name = ''

    def fget(self):
        print "Getting: %s" % self._name
        return self._name

    def fset(self, value):
        print "Setting: %s" % value
        self._name = value.title()

    def fdel(self):
        print "Deleting: %s" %self._name
        del self._name
    name = property(fget, fset, fdel, "I'm the property.")
```

Use this code and see the output:

```
>>> user = Person()
>>> user.name = 'john smith'
Setting: john smith
>>> user.name
Getting: John Smith
'John Smith'
>>> del user.name
Deleting: John Smith
```

Clearly, the result is the same. Note here that `fget`, `fset` and `fdel` methods are optional, but if one is not specified, an exception of `AttributeError` is raised when the respective operation is attempted. For example, a `name` property is declared with `None` as `fset`, and then the developer tries to assign value to `name` attribute. An exception `AttributeError` is raised.

This can be used to define read-only attributes in the system.

```
name = property(fget, None, fdel, "I'm the property")
user.name = 'john smith'
```

Output:

```
Traceback (most recent call last):
File stdin, line 21, in module
user.name = 'john smith'
AttributeError: can't set attribute
```

Creating descriptors using property decorators

Descriptors can be created with Python decorators, as in Listing 4. A Python decorator is a specific change to the Python syntax allowing a more convenient alteration of functions and methods. In this case, attribute management methods are altered. Find more information on the application of Python decorators in the developerWorks article, Decorators make magic easy.

Listing 4. Creating descriptors with property decorators

```
class Person(object):

    def __init__(self):
        self._name = ''

    @property
    def name(self):
        print "Getting: %s" % self._name
        return self._name

    @name.setter
    def name(self, value):
        print "Setting: %s" % value
        self._name = value.title()

    @name.deleter
    def name(self):
        print ">Deleting: %s" % self._name
        del self._name
```

Creating descriptors at run time

All the preceding examples operate with the `name` attribute. The limitation of this approach is the necessity of separately overriding `__set__()`, `__get__()` and `__delete__()` for each attribute. Listing 5 provides a possible solution when a developer wishes to add property attributes at run time. This uses property type to build a data descriptor.

Listing 5. Creating descriptors at run time

```
class Person(object):

    def addProperty(self, attribute):
        # create local setter and getter with a particular attribute name
        getter = lambda self: self._getProperty(attribute)
        setter = lambda self, value: self._setProperty(attribute, value)

        # construct property attribute and add it to the class
        setattr(self.__class__, attribute, property(fget=getter, \
                                                    fset=setter, \
                                                    doc="Auto-generated method"))

    def _setProperty(self, attribute, value):
        print "Setting: %s = %s" %(attribute, value)
        setattr(self, '_' + attribute, value.title())

    def _getProperty(self, attribute):
        print "Getting: %s" %attribute
        return getattr(self, '_' + attribute)
```

Let's play with this code:

```
>>> user = Person()
>>> user.addProperty('name')
>>> user.addProperty('phone')
>>> user.name = 'john smith'
Setting: name = john smith
>>> user.phone = '12345'
Setting: phone = 12345
>>> user.name
Getting: name
'John Smith'
>>> user.__dict__
{'_phone': '12345', '_name': 'John Smith'}
```

This created `name` and `phone` attributes at run time. They are accessible by corresponding name, but they are stored in the object namespace dictionary as _name and _phone, as specified in _setProperty method. Basically, `name` and `phone` are accessors to internal _name and _phone attributes.

You might have one question regarding a _name attribute in the system when the developer tries to add `name` property attribute. The answer is that it will overwrite the existing _name attribute with the new property attribute. This code allows control of how attributes are handled inside a class.

Conclusion

Python descriptors allow for powerful and flexible attribute management with new style classes. Combined with decorators, they make for elegant programming, allowing creation of *Setters* and *Getters*, as well as read-only attributes. It also allows you to run attribute validation on request by value or type. You can apply descriptors in many areas, but use them with discretion to avoid unnecessary code complexity stemming from overriding the normal behaviors of an object.

Related topics

- Python Guide - Invoking Descriptors: Learn to apply descriptors in this section of the Python guide.
- Python Guide - Implementing Descriptors: Explore how to implement descriptors in this section in Python guide.
- Python Properties: Read this explanation of Python properties.
- Decorators make magic easy (David Mertz, developerWorks, December 2006): Look at the Python decorators and their facility for metaprogramming.
- In the developerWorks Linux zone, find hundreds of how-to articles and tutorials, as well as downloads, discussion forums, and a wealth of other resources for Linux developers and administrators.