Python Descriptors, Part 1 of 2

by Marty Alchin on November 23, 2007 about Django

I rather enjoyed writing about a relatively underused feature yesterday, so today is more of the same. Of course, continuing with a focus on Django, today's Python feature is also commonly used throughout a number of Django's internals: descriptors. Python's documentation on descriptors is rather sparse, though there's a great writeup on it already. I won't try to reinvent the wheel in its entirety here, I'll just write up some basic details and how it can be used for Django.

are ırs?

In a nutshell, a descriptor is a way to customize what happens when you reference an attribute on a model. Normally, Python just gets and sets values on attributes without any special processing. It's just basic storage. Sometimes, however, you might want to do more. You might need to validate the value that's being assigned to a value. You may want to retrieve a value and cache it for later use, so that future references don't have all the overhead.

These are all things that would normally need to be done with a method, but if you've already started with a basic attribute, changing to a method would require changing all the code that uses the attribute to use a method call instead. This potential change is a primary motivation for typical Java programs to always use methods even for basic attribute access. The common pattern is to have all attributes private, and provide public access through methods, simply to accommodate potential future changes in the internals of that attribute access.

Python's descriptors as an alternative approach. Instead of starting with methods all the time, you can start with basic attributes and write all the code you want. Then, if you ever need advanced processing to occur when you access those attributes, you can just add in a descriptor to do the work, without updating all the other code. In my opinion, this makes the referring code much cleaner, without having to deal with method calls for attributes. But that's an issue outside the scope of this article.

But the real detail of a descriptor is that it's a Python object that's assigned as an attribute of a class. The object is an instance of a class that's defined in a particular way (described briefly below), and the attribute it's assigned to is the one that will have the special processing. So the actual extra code will be inside the descriptor's class, rather than the class it will be assigned to. This may seem a little weird, but it also makes some sense.

Some examples of descriptors in Django:

- Model managers
- ForeignKey fields (usage)
- ...others that I'm not thinking of right now (sorry, it's late!)

col

A descriptor is implemented as a standard new-style class in Python, and it doesn't need to inherit from anything in particular besides `object`. The real trick to building a descriptor is defining at least one of the following three methods. Note that `instance` below returns to the object where the attribute was accessed, and `owner` is the class where the descriptor was assigned as an attribute.

- `__get__(self, instance, owner)` — This will be called when the attribute is retrieved (`value = obj.attr`), and whatever it returns is what will be given to the code that requested the attribute's value.
- `__set__(self, instance, value)` — This gets called when a value is set to the attribute (`obj.attr = 'value'`), and shouldn't return anything at all.
- `__delete__(self, instance)` — This is called when the attribute is deleted from an object (`del obj.attr`)

Astute readers will quickly notice — and perhaps be confused by — the fact that only `__get__` can receive the owner class, while the rest only receive the instance. Descriptors are assigned to a class, not to an instance, and modifying the class would actually overwrite or delete the descriptor itself, rather than triggering its code. This intentional. Otherwise, once a descriptor is set, it could never be removed or modified without modifying source code and restarting the program. That's not preferable, so only the retrieval method has access to the owner class. It will always be set to the appropriate class, though `instance` may be `None` if the attribute was accessed from the class. This is what Django uses to throw an error if you try to access a manager from an object instead of a class, or a related model (`ForeignKey`) on a class instead of an object.

:he col Also, since descriptors are standard classes that just implement a specific set of methods, they can also contain anything else used on standard Python classes. This is especially useful when defining `__init__` on a descriptor class, so that you can customize descriptors for individual attributes. For example, the following descriptor simulates rolling a die:

```
import random

class Die(object):
    def __init__(self, sides=6):
        self.sides = sides

    def __get__(self, instance, owner):
        return int(random.random() * self.sides) + 1
```

This will default to a 6-sided die, but the number of sides can be explicitly defined on a per-attribute basis. Then, when the attribute is accessed, the `__get__` method will be called, returning a random number based on the number of sides the die was created with.

ors As mentioned above, descriptors are assigned to classes, and the special methods are called automatically when the attribute is accessed, and the method used depends on what type of access is being performed. The `Die` example above might be used as follows:

```
class Game(object):
    d6 = Die()
    d10 = Die(sides=10)
    d20 = Die(sides=20)
```

Then, these special attributes can be accessed either on that class or on any instances of it.

```
>>> Game.d6
5
>>> Game.d10
8
>>> Game.d20
19
>>> Game.d20
3
>>> game = Game()
>>> game.d20
12
```

I think that's enough to cover for one day. Tomorrow's post will go into some more detail, including how descriptors can be used to create highly specialized model fields. Stay tuned!