Python Descriptors, Part 2 of 2

by Marty Alchin on November 24, 2007 about Django

Yesterday, I gave a basic overview of descriptors and what they can do, including a simple example to demonstrate one in action. That's all well and good, but today I'll explain how this can be genuinely useful in your apps, particularly when used in models.

h a
tor

Yesterday's example generated a new value each time it was accessed, which is really only useful in a few situations. More often, you'll need to still store a value somewhere, and just do something special when its modified or retrieved. There are a few ways to approach this, but I'll just cover one.

The simplest way to store a value for a desciptor takes advantage of a subtle distinction of how Python accesses values on an instance object. Every Python object has a namespace that's separate from the namespace of its class, so that each object can have different values attached to it. Normally, the object's attributes are a direct pass-through to this namespace, but descriptors short-circuit that process. Thankfully, Python allows another way to access the object's namespace directly: the `__dict__` attribute of the object.

Every object has a `__dict__` attribute, which is a standard Python dictionary containing mappings for the various values attached to it. Even though descriptors get in the way of how this is normally accessed, your code can use `__dict__` to get at it directly, and it's a great place to store a single value. Yesterday, I mentioned that descriptors can be used to cache values to speed up subsequent accesses, and this is a good way to go about that.

```
from myapp.utils import retrieve

class CachedValue(object):
    def __init__(self, name):
        self.name = name

    def __get__(self, instance, owner):
        if self.name not in instance.__dict__:
            instance.__dict__[self.name] = retrieve()
        return instance.__dict__[self.name]
```

Of course, you'll notice something interesting here. We have to assign the value to the dictionary using a name, and the only way we know what name to use is to supply it explicitly. For this example, the constructor takes a required `name` argument, which will be used for the dictionary's key, but Django provides a much better way to solve this problem. More on that later.

So far, the examples have only involved retrieval. This technique is easily extended to allow the value to be modified as well, by adding a `__set__` method. The following example should look fairly straightforward, now that you know the `__dict__` technique:

```
class SimpleDescriptor(object):
   def __init__(self, name):
        self.name = name

    def __get__(self, instance, owner):
        if self.name not in instance.__dict__:
            raise AttributeError, self.name
        return instance.__dict__[self.name]
    def __set__(self, instance, value):
        instance.__dict__[self.name] = value
```

This also illustrates another interesting point with `__get__`: if the value being retrieved isn't set, the expected behavior is to raise an `AttributeError`. That's what Python does internally, and that's what most code will be expecting when this occurs.

eld

One of the most important uses for descriptors in Django is when creating new `Field` types for use with models. There's a lot that can be done when creating new field types, but that's a topic for its own series of posts, perhaps some other time. For today's purposes, I'll just cover how descriptors can help the process along. Descriptors are especially useful for model fields, because they allow you to integrate specialized Python types with the standard Django database API. *Applicaiton Programming Interface*

Of course, his is all fairly educational. This particular process is done much more easily by Django now, if you're tracking trunk. This information is still good to know, though, because the official support in Django uses descriptors behind the scenes, and not all situations are covered, so you might need to implement this yourself if you find the need.

The base class, `Field`, makes use of a special hook in Django, by defining a method called `contribute_to_class`, and many subclasses override this to provide their own functionality. Again, I won't get into everything that's possible with this, but it provides a very simple solution to our naming problem. Essentially, this method gets called for any object that defines it, instead of being simply attached to the class as normal. The method uses the following definition:

```
def contribute_to_class(self, cls, name)
```

- `self` - the object being assigned to the model
- `cls` - the model class the object is being assigned to
- `name` - the name that was used in the assignment.

That's right, `contribute_to_class` gets the name that was given to the object when it was assigned, so we don't have to expect anyone to provide it explicitly!

ing

This isn't a complete tutorial for subclassing `Field`, just like the last one wasn't a complete discussion of descriptors. There's plently more that can be done, but the best place I can point to is the source for GeoDjango, where Robert Coup so brilliantly implemented descriptors for a very specialized use case. Beyond that, be sure to read the source to Malcolm's recent addition to Django's source, to make this all a lot easier.