Computing Thoughts

Decorators I: Introduction to Python Decorators

by Bruce Eckel
October 18, 2008

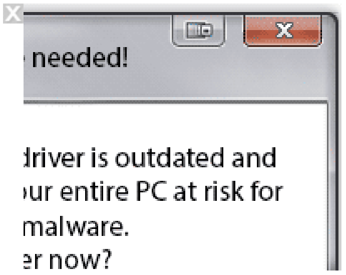
Summary
This amazing feature appeared in the language almost apologetically and with concern that it might not be that useful.

I predict that in time it will be seen as one of the more powerful features in the language. The problem is that all the introductions to decorators that I have seen have been rather confusing, so I will try to rectify that here.

(This series of articles will be incorporated into the open-source book Python 3 Patterns & Idioms).

Decorators vs. the Decorator Pattern

First, you need to understand that the word "decorator" was used with some trepidation, because there was concern that it would be completely confused with the *Decorator* pattern from the Design Patterns book. At one point other terms were considered for the feature, but "decorator" seems to be the one that sticks.



the *Decorator* pattern, but that's an
are best equated to macros.

obably have had experience with C
vere (1) they were in a different
s bizarre, and often inconsistent with

nents

sometimes have to jump through some enormous and untenable hoops, which follows from (2) these annotation features have their hands tied by the bondage-and-discipline (or as Martin Fowler gently puts it: "Directing") nature of those languages.

In a slightly different vein, many C++ programmers (myself included) have noted the generative abilities of C++ templates and have used that feature in a macro-like fashion.

Many other languages have incorporated macros, but without knowing much about it I will go out on a limb and say that Python decorators are similar to Lisp macros in power and possibility.

The Goal of Macros

I think it's safe to say that the goal of macros in a language is to provide a way to modify elements of the language. That's what decorators do in Python -- they modify functions, and in the case of *class decorators*, entire classes. This is why they usually provide a simpler alternative to metaclasses.

The major failings of most language's self-modification approaches are that they are too restrictive and that they require a different language (I'm going to say that Java annotations with all the hoops you must jump through to produce an interesting annotation comprises a "different language").

Python falls into Fowler's category of "enabling" languages, so if you want to do modifications, why create a different or restricted language? Why not just use Python itself? And that's what Python decorators do.

What Can You Do With Decorators?

Decorators allow you to inject or modify code in functions or classes. Sounds a bit like *Aspect-Oriented Programming* (AOP) in Java, doesn't it? Except that it's both much simpler and (as a result) much more powerful. For example, suppose you'd like to do something at the entry and exit points of a function (such as perform some kind of security, tracing, locking, etc. -- all the standard arguments for AOP). With decorators, it looks like this:

```
@entryExit
def func1():
    print "inside func1()"

@entryExit
def func2():
    print "inside func2()"
```

The `@` indicates the application of the decorator.

Function Decorators

A function decorator is applied to a function definition by placing it on the line before that function definition begins. For example:

```
@myDecorator
def aFunction():
    print "inside aFunction"
```

When the compiler passes over this code, `aFunction()` is compiled and the resulting function object is passed to the `myDecorator` code, which does something to produce a function-like object that is then substituted for the original `aFunction()`.

What does the `myDecorator` code look like? Well, most introductory examples show this as a function, but I've found that it's easier to start understanding decorators by using classes as decoration mechanisms instead of functions. In addition, it's more powerful.

The only constraint upon the object returned by the decorator is that it can be used as a function -- which basically means it must be callable. Thus, any classes we use as decorators must implement `__call__`.

What should the decorator do? Well, it can do anything but usually you expect the original function code to be used at some point. This is not required, however:

```
class myDecorator(object):

    def __init__(self, f):
        print "inside myDecorator.__init__()"
        f() # Prove that function definition has completed

    def __call__(self):
        print "inside myDecorator.__call__()"

@myDecorator
def aFunction():
    print "inside aFunction()"
print "Finished decorating aFunction()"

aFunction()
```

When you run this code, you see:

```
inside myDecorator.__init__()
inside aFunction()
Finished decorating aFunction()
inside myDecorator.__call__()
```

Notice that the constructor for `myDecorator` is executed at the point of decoration of the function. Since we can call `f()` inside `__init__()`, it shows that the creation of `f()` is complete before the decorator is called. Note also that the decorator constructor receives the function object being decorated. Typically, you'll capture the function object in the constructor and later use it in the `__call__()` method (the fact that decoration and calling are two clear phases when using classes is why I argue that it's easier and more powerful this way).

When `aFunction()` is called after it has been decorated, we get completely different behavior; the `myDecorator.__call__()` method is called instead of the original code. That's because the act of decoration *replaces* the original function object with the result of the decoration -- in our case, the `myDecorator` object replaces `aFunction`. Indeed, before decorators were added you had to do something much less elegant to achieve the same thing:

```
def foo(): pass
foo = staticmethod(foo)
```

With the addition of the `@` decoration operator, you now get the same result by saying:

```
@staticmethod
def foo(): pass
```

This is the reason why people argued against decorators, because the `@` is just a little syntax sugar meaning "pass a function object through another function and assign the result to the original function."

The reason I think decorators will have such a big impact is because this little bit of syntax sugar changes the way you think about programming. Indeed, it brings the idea of "applying code to other code" (i.e.: macros) into mainstream thinking by formalizing it as a language construct.

Slightly More Useful

Now let's go back and implement the first example. Here, we'll do the more typical thing and actually use the code in the decorated functions:

```
class entryExit(object):

    def __init__(self, f):
        self.f = f

    def __call__(self):
        print "Entering", self.f.__name__
        self.f()
        print "Exited", self.f.__name__

@entryExit
def func1():
    print "inside func1()"

@entryExit
def func2():
    print "inside func2()"

func1()
func2()
```

The output is:

```
Entering func1
inside func1()
Exited func1
Entering func2
inside func2()
Exited func2
```

You can see that the decorated functions now have the "Entering" and "Exited" trace statements around the call.

The constructor stores the argument, which is the function object. In the call, we use the `__name__` attribute of the function to display that function's name, then call the function itself.

Using Functions as Decorators

The only constraint on the result of a decorator is that it be callable, so it can properly replace the decorated function. In the above examples, I've replaced the original function with an object of a class that has a `__call__()` method. But a function object is also callable, so we can rewrite the previous example using a function instead of a class, like this:

```
def entryExit(f):
    def new_f():
        print "Entering", f.__name__
        f()
        print "Exited", f.__name__
    return new_f

@entryExit
def func1():
    print "inside func1()"

@entryExit
def func2():
    print "inside func2()"

func1()
func2()
print func1.__name__
```

`new_f()` is defined within the body of `entryExit()`, so it is created and returned when `entryExit()` is called. Note that `new_f()` is a *closure*, because it captures the actual value of `f`.

Once `new_f()` has been defined, it is returned from `entryExit()` so that the decorator mechanism can assign the result as the decorated function.

The output of the line `print func1.__name__` is `new_f`, because the `new_f` function has been substituted for the original function during decoration. If this is a problem you can change the name of the decorator function before you return it:

```
def entryExit(f):
    def new_f():
        print "Entering", f.__name__
        f()
        print "Exited", f.__name__
    new_f.__name__ = f.__name__
    return new_f
```

The information you can dynamically get about functions, and the modifications you can make to those functions, are quite powerful in Python.

More Examples

Now that you have the basics, you can look at some more examples of decorators here. Note the number of these examples that use classes rather than functions as decorators.

In this article I have intentionally avoided dealing with the arguments of the decorated function, which I will look at in the next article.

Talk Back!

Have an opinion? Readers have already posted 34 comments about this weblog entry. Why not add yours?

RSS Feed

If you'd like to be notified whenever Bruce Eckel adds a new entry to his weblog, subscribe to his RSS feed.

Digg | del.icio.us | Reddit

About the Blogger

Bruce Eckel (www.BruceEckel.com) provides development assistance in Python with user interfaces in Flex. He is the author of Thinking in Java (Prentice-Hall, 1998, 2nd Edition, 2000, 3rd Edition, 2003, 4th Edition, 2005), the Hands-On Java Seminar CD ROM (available on the Web site), Thinking in C++ (PH 1995; 2nd edition 2000, Volume 2 with Chuck Allison, 2003), C++ Inside & Out (Osborne/McGraw-Hill 1993), among others. He's given hundreds of presentations throughout the world, published over 150 articles in numerous magazines, was a founding member of the ANSI/ISO C++ committee and speaks regularly at conferences.

Google [Search]

( ) Web (●) Artima.com

innoApp Advertisement