Computing Thoughts

Python Decorators II: Decorator Arguments

by Bruce Eckel

October 19, 2008

Summary

The decorator mechanism behaves quite differently when you pass arguments to the decorator.

(This is part II of a series that will become a chapter in Python 3 Patterns & Idioms; you can find part I here).

Review: Decorators without Arguments

In part I, I showed how to use decorators without arguments, primarily using classes as decorators because I find them easier to think about.

If we create a decorator without arguments, the function to be decorated is passed to the constructor, and the `__call__()` method is called whenever the decorated function is invoked:

```
class decoratorWithoutArguments(object):

    def __init__(self, f):
        """
        If there are no decorator arguments, the function
        to be decorated is passed to the constructor.
        """
        print "Inside __init__()"
        self.f = f

    def __call__(self, *args):
        """
        The __call__ method is not called until the
        decorated function is called.
        """
        print "Inside __call__()"
        self.f(*args)
        print "After self.f(*args)"

@decoratorWithoutArguments
def sayHello(a1, a2, a3, a4):
    print 'sayHello arguments:', a1, a2, a3, a4

print "After decoration"

print "Preparing to call sayHello()"
sayHello("say", "hello", "argument", "list")
print "After first sayHello() call"
```

```
sayHello("a", "different", "set of", "arguments")
print "After second sayHello() call"
```

Any arguments for the decorated function are just passed to `__call__()`. The output is:

```
Inside __init__()
After decoration
Preparing to call sayHello()
Inside __call__()
sayHello arguments: say hello argument list
After self.f(*args)
After first sayHello() call
Inside __call__()
sayHello arguments: a different set of arguments
After self.f(*args)
After second sayHello() call
```

Notice that `__init__()` is the only method called to perform decoration, and `__call__()` is called every time you call the decorated `sayHello()`.

Decorators with Arguments

Now let's modify the above example to see what happens when we add arguments to the decorator:

```
class decoratorWithArguments(object):

    def __init__(self, arg1, arg2, arg3):
        """
        If there are decorator arguments, the function
        to be decorated is not passed to the constructor!
        """
        print "Inside __init__()"
        self.arg1 = arg1
        self.arg2 = arg2
        self.arg3 = arg3

    def __call__(self, f):
        """
        If there are decorator arguments, __call__() is only called
        once, as part of the decoration process! You can only give
        it a single argument, which is the function object.
        """
        print "Inside __call__()"
        def wrapped_f(*args):
            print "Inside wrapped_f()"
            print "Decorator arguments:", self.arg1, self.arg2, self.arg3
            f(*args)
            print "After f(*args)"
        return wrapped_f

@decoratorWithArguments("hello", "world", 42)
def sayHello(a1, a2, a3, a4):
    print 'sayHello arguments:', a1, a2, a3, a4

print "After decoration"

print "Preparing to call sayHello()"
sayHello("say", "hello", "argument", "list")
print "after first sayHello() call"
sayHello("a", "different", "set of", "arguments")
print "after second sayHello() call"
```

From the output, we can see that the behavior changes quite significantly:

```
Inside __init__()
Inside __call__()
After decoration
Preparing to call sayHello()
Inside wrapped_f()
Decorator arguments: hello world 42
sayHello arguments: say hello argument list
After f(*args)
after first sayHello() call
Inside wrapped_f()
Decorator arguments: hello world 42
sayHello arguments: a different set of arguments
After f(*args)
after second sayHello() call
```

Now the process of decoration calls the constructor and then immediately invokes `__call__()`, which can only take a single argument (the function object) and must return the decorated function object that replaces the original. Notice that `__call__()` is now only invoked once, during decoration, and after that the decorated function that you return from `__call__()` is used for the actual calls.

Although this behavior makes sense -- the constructor is now used to capture the decorator arguments, but the object `__call__()` can no longer be used as the decorated function call, so you must instead use `__call__()` to perform the decoration -- it is nonetheless surprising the first time you see it because it's acting so much differently than the no-argument case, and you must code the decorator very differently from the no-argument case.

Decorator Functions with Decorator Arguments

Finally, let's look at the more complex decorator function implementation, where you have to do everything all at once:

```
def decoratorFunctionWithArguments(arg1, arg2, arg3):
    def wrap(f):
        print "Inside wrap()"
        def wrapped_f(*args):
            print "Inside wrapped_f()"
            print "Decorator arguments:", arg1, arg2, arg3
            f(*args)
            print "After f(*args)"
        return wrapped_f
    return wrap

@decoratorFunctionWithArguments("hello", "world", 42)
def sayHello(a1, a2, a3, a4):
    print 'sayHello arguments:', a1, a2, a3, a4

print "After decoration"

print "Preparing to call sayHello()"
sayHello("say", "hello", "argument", "list")
print "after first sayHello() call"
sayHello("a", "different", "set of", "arguments")
print "after second sayHello() call"
```

Here's the output:

```
Inside wrap()
After decoration
Preparing to call sayHello()
Inside wrapped_f()
Decorator arguments: hello world 42
```

```
sayHello arguments: say hello argument list
After f(*args)
after first sayHello() call
Inside wrapped_f()
Decorator arguments: hello world 42
sayHello arguments: a different set of arguments
After f(*args)
after second sayHello() call
```

The return value of the decorator function must be a function used to wrap the function to be decorated. That is, Python will take the returned function and call it at decoration time, passing the function to be decorated. That's why we have three levels of functions; the inner one is the actual replacement function.

Because of closures, `wrapped_f()` has access to the decorator arguments `arg1`, `arg2` and `arg3`, *without* having to explicitly store them as in the class version. However, this is a case where I find "explicit is better than implicit," so even though the function version is more succinct I find the class version easier to understand and thus to modify and maintain.

Next

In the next installment I'll show a practical example of decorators -- a build system created atop Python -- and in the final installment we'll look at class decorators.

Talk Back!

Have an opinion? Readers have already posted 19 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.

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.