January 2, 2013

Decorators and Functional Python

Decorators are one of Python's great features. In addition to their intrinsic usefulness in the language, they also help us to think in an interesting way — a *functional* way.

I intend to explain how decorators work from the ground up. We'll start by covering a few topics you'll need in order to understand decorators. After that, we'll dive in and explore a few simple decorators and how they work. Finally, we'll talk about some more advanced ways to use decorators, such as passing them optional arguments or chaining them together.

First, let's define what a Python function is in the simplest way I can think of. From that simple definition, we can then define decorators in a similarly simple way.

> *__A function__ is a block of reusable code that performs a specific task.*

Okay, so then what is a decorator?

> *__A decorator__ is a function that modifies other functions.*

Now let's start to expand on that definition of decorators, starting with a couple prerequisite explanations.

Functions are first class objects

In Python, everything is an object. What this means is that functions can be referred to by name and passed around like any other object. For example:

```
def traveling_function():
    print "Here I am!"
```

```
function_dict = {
    "func": traveling_function
}

trav_func = function_dict['func']
trav_func()
# >> Here I am!
```

`traveling_function` was assigned as the value of the `func` key in the `function_dict` dictionary and can still be called like normal.

First class functions allow for higher order functions

We can pass functions around like any other object. We can pass them as values to dictionaries, put them in lists, or assign them as object properies. So couldn't we pass them as arguments to another function? We can! A function that accepts another function as a parameter or returns another function is called a *higher order function*.

```
def self_absorbed_function():
    return "I'm an amazing function!"


def printer(func):
    print "The function passed to me says: " + func()

# Call `printer` and give it `self_absorbed_function` as an argument
printer(self_absorbed_function)
# >> The function passed to me says: I'm an amazing function!
```

So here you can see that a function can be passed to another function as an argument, and that function can then invoke the passed in function. This allows us to create some interesting functions, like decorators!

The basics of a decorator

At heart, a decorator is just a function that takes another function as an argument. In most cases they return a function that is a modified version of the function they are wrapping. Let's look at the simplest decorator we can that might help us understand how all of this works — the identity decorator.

```
def identity_decorator(func):
    def wrapper():
        func()
    return wrapper


def a_function():
    print "I'm a normal function."

# `decorated_function` is the function that `identity_decorator` returns, which
# is the nested function, `wrapper`
decorated_function = identity_decorator(a_function)

# This calls the function that `identity_decorator` returned
decorated_function()
# >> I'm a normal function
```

Here, `identity_decorator` does not modify the function it wraps at all. It simply returns a function (`wrapper`) that when called, will invoke the original function `identity_decorator` received as an argument. This is a useless decorator!

What's interesting about `identity_decorator` is that `wrapper` has access to the `func` variable even though `func` was not passed in as one of its arguments. This is due to closures.

Closures

> *Closure is a fancy term meaning that when a function is declared, it maintains a reference to the lexical environment in which it was declared.*

When `wrapper` was defined in the previous example, it had access to the `func` variable in its local scope. This means that throughout the life of `wrapper` (which got returned and assigned to the name `decorated_function`), it will have access to the `func` variable. Once `identity_decorator` returns, the only way to access `func` is through `decorated_function`. `func` does not exist as a variable anywhere other than inside `decorated_function`'s closure environment.

A simple decorator

Now let's create a decorator that will actually be of a little use. All this decorator will do is log how many times the function it modifies gets called.

```
def logging_decorator(func):
    def wrapper():
        wrapper.count += 1
        print "The function I modify has been called {0} times(s).".format(
              wrapper.count)
        func()
    wrapper.count = 0
    return wrapper


def a_function():
    print "I'm a normal function."

modified_function = logging_decorator(a_function)

modified_function()
# >> The function I modify has been called 1 time(s).
# >> I'm a normal function.

modified_function()
# >> The function I modify has been called 2 time(s).
# >> I'm a normal function.
```

We said a decorator modifies a function, and it can be useful to think of it like that. But as you can see in our example, what `logging_decorator` does is return a *new* function that is similar to `a_function`, but with the addition of a logging feature.

In this example, `logging_decorator` not only accepts a function as a parameter, it also returns a function, `wrapper`. Each time the function that `logging_decorator` returns gets called, it increments `wrapper.count`, prints it, and then calls the function that `logging_decorator` is wrapping.

You might be wondering why our counter is a property of `wrapper` instead of a regular variable. Wouldn't `wrapper`'s closure environment give us access to any variable declared in its local scope? Yes, but there's a catch. In Python, a closure provides full *read* access to any variable in the function's scope chain, but only provides *write* access to mutable objects (lists, dictionaries, etc.). An integer is an immutable object in Python, so we wouldn't be able to increment its value inside of `wrapper`. Instead, we made our counter a property of `wrapper`, a mutable object, and thus we can increment it all we like!

Decorator syntax

In the last example, we saw that a decorator can be used by passing it a function as an argument, thus 'wrapping' that function with the decorator function. However, Python also has a syntax pattern that makes this more intuitive and easier to read once you are comfortable with decorators.

```
# In the previous example, we used our decorator function by passing the
# function we wanted to modify to it, and assigning the result to a variable

def some_function():
    print "I'm happiest when decorated."

# Here we will make the assigned variable the same name as the wrapped function
some_function = logging_decorator(some_function)
```

```
# We can achieve the exact same thing with this syntax:

@logging_decorator
def some_function():
    print "I'm happiest when decorated."
```

Using the decorator syntax, this is the bird's eye view of what happens:

1. The interpreter reaches the decorated function, compiles `some_function`, and gives it the name 'some_function'.
2. That function is then passed to the decorator function that is named in the decoration line (`logging_decorator`).
3. The return value of the decorator function (usually another function that wraps the original) is substituted for the original function (`some_function`). It is now bound to the name 'some_function'.

With these steps in mind, let's annotate the `identity_decorator` a little for clarification.

```
def identity_decorator(func):
    # Everything here happens when the decorator LOADS and is passed
    # the function as described in step 2 above
    def wrapper():
        # Things here happen each time the final wrapped function gets CALLED
        func()
    return wrapper
```

Hopefully those comments are instructive. Only commands that are inside of the function that the decorator returns get called each time the wrapped function is invoked. Commands written outside of that return function will only happen once — when the decorator first gets passed its wrapped function in step 2 described above.

There's one more thing I'd like to explain a little before we start looking at some more interesting decorators.

*args and **kwargs

You may have seen these sometimes confusing brothers before. Let's talk about them one at a time.

- A python function can accept a variable number of positional arguments by using the `*args` syntax in its parameter list. `*args` will combine all non-keyword arguments into a single tuple of arguments that can be accessed within the function. Conversely, when `*args` is used in the argument list of a function invocation, it will expand a tuple of arguments out into a series of positional arguments.

```
def function_with_many_arguments(*args):
    print args

# `args` within the function will be a tuple of any arguments we pass
# which can be used within the function like any other tuple
function_with_many_arguments('hello', 123, True)
# >> ('hello', 123, True)
```

```
def function_with_3_parameters(num, boolean, string):
    print "num is " + str(num)
    print "boolean is " + str(boolean)
    print "string is " + string

arg_list = [1, False, 'decorators']

# arg_list will be expanded into 3 positional arguments by the `*` symbol
function_with_3_parameters(*arg_list)
# >> num is 1
# >> boolean is False
# >> string is decorators
```

To reiterate: in a parameter list, `*args` will *condense* a series of arguments into one tuple named 'args', in an argument list, `*args` will *expand* an iterable of arguments into a series of positional arguments it applies to the function.

As you saw in the argument expansion example, the `*` symbol can be used with names other than 'args'. It is just convention to use the form `*args` when condensing/expanding generic argument lists.

- `**kwargs` behaves similarly to its brother, `*args`, but it works with keyword arguments instead of positional. If `**kwargs` is used in a function parameter list, it will collect as many extra keyword arguments the function has received and place them into a dictionary. If used in a function argument list, it will expand a dictionary into a series of keyword arguments.

```
def function_with_many_keyword_args(**kwargs):
    print kwargs

function_with_many_keyword_args(a='apples', b='bananas', c='cantalopes')
# >> {'a': 'apples', 'b': 'bananas', 'c': 'cantalopes'}
```

```
def multiply_name(count=0, name=''):
    print name * count

arg_dict = {'count': 3, 'name': 'Brian'}

multiply_name(**arg_dict)
# >> BrianBrianBrian
```

Now that you understand how `*args` and `**kwargs` work their magic, let's move on to studying a decorator you just might find useful.

Memoization

Memoization is a way to avoid repeating potentially expensive calculations. You do this by caching the result of a function each time it runs. This way, the next time the function runs with the same arguments, it will return the result from the cache, not having to calculate the result an additional time.

```
from functools import wraps


def memoize(func):
    cache = {}
    @wraps(func)
    def wrapper(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]
    return wrapper

@memoize
def an_expensive_function(arg1, arg2, arg3):
    ...
```

You probably noticed a strange `@wraps` decorator in this code sample. I'll explain that briefly before we talk about `memoize` as a whole a little more.

- A side effect of using decorators is that the function that gets wrapped loses it's natural `__name__`, `__doc__`, and `__module__` attributes. The `wraps` function is used as a *decorator* that wraps the function that a decorator returns, restoring those three attributes to the values they would have if the wrapped function was not decorated. For instance: `an_expensive_function`'s name (as seen by `an_expensive_function.__name__`) would have been 'wrapper' if we did not use the `wraps` decorator.

I think `memoize` represents a good use case for decorators. It serves a purpose that will be desired in a lot of functions, and by creating it as a generic decorator, we can add its functionality to any function that can benefit from it. This avoids the need to reproduce this functionality in many different places. By not repeating ourselves it makes our code easier to maintain, and also easier to read and understand. You instantly understand that the function is memoized by reading a single word.

I should note that memoization is only appropriate to use on pure functions. That is a function that is guaranteed to always produce the same result given a certain set of arguments. If it is dependent upon global variables not passed as arguments, I/O, or anything else that might affect the return value, memoization will produce confusing results! Also, a pure function does not have any side effects. So if your function increments a counter, or calls a method on another object, or anything else that isn't represented in the return value the function produces, that side effect will not be performed when the result is returned by the cache.

Class decorators

Originally, we said a decorator is a function that modifies another function, but they can also be used to modify classes or methods. It is not as common to decorate classes, but it can be useful tool in certain instances as an alternative to metaclasses.

```
foo = ['important', 'foo', 'stuff']


def add_foo(klass):
    klass.foo = foo
    return klass


@add_foo
class Person(object):
    pass

brian = Person()

print brian.foo
# >> ['important', 'foo', 'stuff']
```

Now any objects of class `Person` will have the super-important `foo` attribute! Notice that since we're decorating a class, our decorator doesn't return a function, but naturally, a class. Let's update our decorator definition:

> *A **decorator** is a function that modifies functions, methods or classes.*

Decorators as classes

It turns out I withheld something else from you earlier. Not only can a decorator decorate a class, a decorator can *be* a class! The only requirement of a decorator is that its return value must be callable. This means that it must implement the `__call__` magic method, which gets called behind the scenes when you *call* an object. Functions set this method implicitly of course. Let's recreate the `identity_decorator` as a class to see how this works.

```
class IdentityDecorator(object):
    def __init__(self, func):
        self.func = func

    def __call__(self):
        self.func()


@IdentityDecorator
```

```
def a_function():
    print "I'm a normal function."

a_function()
# >> I'm a normal function
```

Here is what happens in this example:

- When `IdentityDecorator` decorates `a_function`, it behaves just like a decorator that is a function. This snippit would be equivalent to the example's decoration syntax: `a_function = IdentityDecorator(a_function)`. The class decorator gets called (thus *instantiated*) with the function it decorates passed to it as an argument.
- When `IdentityDecorator` is instantiated its initialization function, `__init__`, gets called with the decorated function passed as an argument. In this case all it does is assign that function to an attribute so it can be accessed later by other methods.
- Finally, when `a_function` (which is really the returned `IdentityDecorator` object wrapping `a_function`) gets called, the object's `__call__` method gets invoked. Since this is just an identity decorator, it simply calls the function it decorates.

Let's update our definition of a decorator once more!

> ***A decorator*** *is a callable that modifies functions, methods or classes.*

Decorators with arguments

Sometimes you need to change your decorator's behavior on a case by case basis. You can do that by passing arguments.

```
from functools import wraps


def argumentative_decorator(gift):
    def func_wrapper(func):
        @wraps(func)
        def returned_wrapper(*args, **kwargs):
            print "I don't like this " + gift + " you gave me!"
            return func(gift, *args, **kwargs)
        return returned_wrapper
    return func_wrapper
```

```
@argumentative_decorator("sweater")
def grateful_function(gift):
    print "I love the " + gift + "! Thank you!"

grateful_function()
# >> I don't like this sweater you gave me!
# >> I love the sweater! Thank you!
```

Let's take a look at how this decorator function would work if we didn't use decorator syntax:

```
# If we tried to invoke without an argument:
grateful_function = argumentative_function(grateful_function)

# But when given an argument, the pattern changes to:
grateful_function = argumentative_decorator("sweater")(grateful_function)
```

The main thing to notice is that when given arguments, a decorator is first invoked with only those arguments — the wrapped function is not one of them like normal. After that function call returns, the function the decorator is wrapping is passed to the function that was *returned* by the initial invocation of the decorator with its arguments (in this case, the return value of: `argumentative_decorator("sweater")`).

Step by step:

1. The interpreter reaches the decorated function, compiles `grateful_function`, and binds it to the name 'grateful_function'.
2. `argumentative_decorator` is called, and passed the argument "sweater". It returns `func_wrapper`.
3. `func_wrapper` is invoked with `grateful_function` as an argument. `func_wrapper` returns `returned_wrapper`.
4. Finally, `returned_wrapper` is substituted for the original function, `grateful_function`, and is thus bound to the name 'grateful_function'.

I think this line of events is a little harder to follow than when there are no decorator arguments, but if you take some time to think it through, hopefully it will make sense.

Decorators with optional arguments

There are many ways to accept optional arguments with decorators. Depending on if you want to use positional arguments, keyword arguments, or both, you will have to use a slightly different pattern. I'll show one way to accept an optional keyword argument:

```
from functools import wraps

GLOBAL_NAME = "Brian"


def print_name(function=None, name=GLOBAL_NAME):
    def actual_decorator(function):
        @wraps(function)
        def returned_func(*args, **kwargs):
            print "My name is " + name
            return function(*args, **kwargs)
        return returned_func

    if not function:  # User passed in a name argument
        def waiting_for_func(function):
            return actual_decorator(function)
        return waiting_for_func

    else:
        return actual_decorator(function)


@print_name
def a_function():
    print "I like that name!"


@print_name(name='Matt')
def another_function():
    print "Hey, that's new!"

a_function()
# >> My name is Brian
# >> I like that name!

another_function()
# >> My name is Matt
# >> Hey, that's new!
```

If we pass the keyword argument `name` to `print_name` it will behave similarly to `argumentative_decorator` in the previous example. That is, first `print_name` will be called with `name` as its argument. Then the function that first invocation returned will be passed the function it is wrapping.

If we don't provide a `name` argument, `print_name` will behave like the argument-less decorators we've seen in the past. It will just get invoked with the function its wrapping as the sole argument.

`print_name` accounts for both possibilities. It checks to see if it received the wrapped function as an argument. If not, it returns a function `waiting_for_func` that will get invoked with the wrapped function as its argument. If it did receive the function as an argument, it skips that intermediary step and just invokes the `actual_decorator` immediately.

Chaining decorators

Let's explore one last feature of decorators today: chaining. You can stack more than one decorator on any given function. This can be used to construct functions in a way that is similar to how multiple inheritance can be used to construct classes. It is probably best to avoid going crazy with this though.

```
@print_name('Sam')
@logging_decorator
def some_function():
    print "I'm the wrapped function!"

some_function()
# >> My name is Sam
# >> The function I modify has been called 1 time(s).
# >> I'm the wrapped function!
```

When you chain decorators, the order in which they are stacked is bottom to top. The function that is being wrapped, `some_function`, is compiled and passed to the first decorator above it (`logging_decorator`). Then the return value of that first decorator is passed to the second. And so it will continue for each decorator in the chain.

Since both of the decorators we used here `print` a value and then run the function they were passed, this means the last decorator in the chain, `print_name`, will print the first line of output when the wrapped function is called.

Conclusion

I think one of the largest benefits of decorators is that they allow you to think at a slightly higher level of abstraction. If you begin to read a function definition and see that it has a `memoize` decorator, you will instantly understand that you're looking at a memoized function. If the memoization code were included inside of the function body it would require extra mental parsing, and introduce possible misunderstanding. Using decorators also allows for code reuse, which can save time, ease debugging, and make refactoring easier.

Playing with decorators is also a great way to learn about functional concepts like higher order functions and closures.

I hope this has been an enjoyable and informative read!