Intro to Decorators in Python

@EdmontonPy

Matthew Darling

August 12th, 2013

Preface

Fair warning: I may omit details/tell white lies (at first)

Also, pretty much all my examples are barely-modified versions of stuff I found online - the original links should all be in my prep file under the "Content notes" section

What a decorator is

- Decorators are functions that modify other functions (white lie: you can actually use any callable, ie classes - see the link to Python Patterns, Recipes, and Idioms)
- You can use them on your own functions, and on functions from an imported module

The long way

When you're using a decorator, what you're really doing is this (where cache is a decorator that implements, well, caching):

```
function = cache(function)
```

How to use a decorator that someone else wrote

Since there are a lot of good decorators available as PyPI packages or code snippets of dubious quality, you can get a lot of cool stuff without ever writing your own decorators.

So how do you use them?

```
@cache
def function():
    return really_complex_result()
```

And you're done! Pat yourself on the back

OOP examples

- @classmethod
- @staticmethod
- @abc.abstractmethod in Python 3+
- @properties (a transparent alternative to getters/setters - take that, Java!)

How to decorate a function with no arguments

```
def verbose(function):
    print("I'm the decorator - my argument must be a function")
    def wrapper():
        print("This is the wrapper - it calls the original function")
        print("You called", function.__name__)
        result = function()
        print("Now we return the result of the function call")
        return result
    return wrapper

def hello():
    print("Hello")

print("Before we call the decorator, we've defined", hello.__name__)
hello = verbose(hello)

hello()
print("After calling the decorator, the function's name is", hello.__name__)
```

Output:

```
Before we call the decorator, we've defined hello
I'm the decorator - my argument must be a function
This is the wrapper - it calls the original function
You called hello
Hello
Now we return the result of the function call
After calling the decorator, the function's name is wrapper
```

How to decorate a function with known arguments

If you know exactly how many arguments your function takes, you can hardcode the number of arguments for the wrapper function:

```
def elementwise(function):
    def wrapper(arg):
        if hasattr(arg,'__getitem__'):  #is a sequence
            return type(arg)(map(function, arg))
        else:
            #Note that wrapper receives the arguments meant for "function"
            #If "function" required more than one argument, this wouldn't work
            return function(arg)
    return wrapper

@elementwise
def compute(x):
    return x**3 - 1

print(compute(5))
print(compute([1,2,3])) #passing a list
print(compute((1,2,3))) #passing a tuple
```

Output:

```
124
[0, 7, 26]
(0, 7, 26)
```

How to decorate a function with unknown arguments

But if you want your decorator to be more general, you need to support any possible combination of arguments:

```
import time
def benchmark(function):
    """A decorator that prints the time a function takes to execute."""
    def wrapper(*args, **kwargs): #This function will accept any arguments
        t = time.clock()
        result = function(*args, **kwargs)
        print("The function", function.__name__, "took", time.clock()-t)
        return result
    return wrapper

@benchmark
def waste_time(wait, test="nothing", extra="Read all about it!"):
    time.sleep(wait)
    print("Experimenting with:", test)
    print("Breaking news:", extra)
```

Testing

```
waste_time(3)
waste_time(3, test="decorators")
waste_time(3, extra="this is best presentation I've seen all day")
```

Output:

```
Experimenting with: nothing
Breaking news: Read all about it!
The function waste_time took 2.99943593545
Experimenting with: decorators
Breaking news: Read all about it!
The function waste_time took 2.99978290028
Experimenting with: nothing
Breaking news: this is best presentation I've seen all day
The function waste_time took 2.9998539511
```

How to write a decorator factory

A decorator that takes arguments arguments is sometimes called a "decorator factory". When you see

```
@decorator(argument)
```

read it as: "wrap the following function with the output of the decorator factory"

Here's how it works:

```
#call example with the return value of test
example(test("this is a test"))
#call the return value of test_factory with "this is a test"
test_factory(args=[])("this is a test")
#Factory returns a function
#Call its return value with "this is a test"
```

Similar to:

```
decorator_factory(argument)(function)
#Call decorator_factory(argument), then call its return value with function
```

Very simple decorator factory with Flask

Courtesy of this blog post, here's a simple example:

```
from Flask import flask
app = Flask(__name__)

#the app.route decorator has an argument
#technically, you could call it a decorator factory
@app.route('/')
def index():
    return "Hello, EdmontonPy!"

if __name__ == "__main__":
    app.run(debug = True) #we have no main function - we delegate to Flask
```

Real example of a decorator factory

```
def deprecated(replacement=None):
    print("You've called the deprecated factory with", replacement.__name__
          if replacement else None)
    def decorator(old_function):
        print("The decorator function received", old_function.__name__,
              "as its sole argument")
        def wrapper(*args, **kwargs):
            msg = "{} is deprecated".format(old_function.__name__)
            if replacement is not None:
                msg += "; use {} instead".format(replacement.__name__)
                print(msg)
                return replacement(*args, **kwargs)
            else:
                return old_function(*args, **kwargs)
        return wrapper
    return decorator
```

Example usage

```
print("Calling the factory with no arguments")
test = deprecated()

def sum_many(*args):
    return sum(args)

print("Calling the factory with a replacement function")
many_deprecated = deprecated(sum_many)
print("The factory returned",
      many_deprecated.__name__)

#Equivalent: @many_deprecated
#def sum_couple ..etc..
@deprecated(sum_many)
def sum_couple(a, b):
    return a + b

print("Going to call sum_couple now")
print(sum_couple(2, 2))
```

Output:

```
Calling the factory with no arguments
You've called the deprecated factory with None
Calling the factory with a replacement function
You've called the deprecated factory with sum_many
The factory returned decorator
You've called the deprecated factory with sum_many
The decorator function received sum_couple as its sole argument
Going to call sum_couple now
sum_couple is deprecated; use sum_many instead
4
```

functools.wraps and the decorator module

Remember how we saw "After calling the decorator, the function's name is wrapper"?

You'll never be able to debug that, because every decorated function will have a __name__ of "wrapper"

Solutions: functools.wraps, or the decorator module

- functools.wraps is lightweight and does the most important things
- The decorator module offers a bit of extra functionality (check the docs)
- But which you use is more a question of personal/aesthetic preference

functools example

```
from functools import wraps

def verbose(function):
    print("I'm the decorator - I can only take one argument")
    @wraps(function)
    def wrapper(*args, **kwargs):
        print("This is your wrapper speaking")
        result = function(*args, **kwargs)
        return result
    return wrapper

@verbose
def hello():
    print("Hello")

hello()
```

Output:

```
I'm the decorator - I can only take one argument
This is your wrapper speaking
Hello
```

decorator module example

```
from decorator import decorator

@decorator
def verbose(function, *args, **kwargs):
    print("I'm the wrapper")
    result = function(*args, **kwargs)
    return result

@verbose
def hello():
    print("Hello")

hello()
```

Output:

```
I'm the wrapper
Hello
```

Decorators are often complicated

Chris McDonough, author of Pyramid, thinks that there are often simpler ways to accomplish what decorators do - namely, including the same code in the body of your function

As I see it, there are three criteria you can think about:

- Whether the decorator provides core behaviour to the function
- Whether you're writing a "framework" (defined soon)
- Whether you're trying to fix someone else's function(s)

How important is the decorator?

Whether you should use a decorator might depend on how crucial the behaviour of the decorator is to what the function does

Example: If you always want to do some logging in a function, put it in the function.

If you're turning on logging temporarily, or it's optional - then a decorator you can disable makes sense.

Decorators in frameworks

Decorators are good for "frameworks" - eg web frameworks like Flask, Django, and command line frameworks like Aaargh - where the decorator executes the user's code

In short, rather than having your own main()-like function, when you're using a framework you use **their** main()-like function

It then executes your decorated functions based on how you've configured it - see the next two examples

Web frameworks

```
from Flask import flask
app = Flask(__name__)

#when someone visits "http://www.examplesite.com/",
#they'll see "Hello, EdmontonPy!"
@app.route('/')
def index():
    return "Hello, EdmontonPy!"

if __name__ == "__main__":
    app.run(debug = True) #we have no main function - we delegate to Flask
```

Command line programs

```
from __future__ import print_function
import aaargh
app = aaargh.App(description="A simple greeting application.")

#if the program is called with "hello" as an argument, this function is called
@app.cmd
def hello():
    print("Hello, EdmontonPy!", end="")

if __name__ == "__main__":
    app.run(["hello"]) #again, we're delegating to aaargh - though I'm faking a command line argument
```

Output:

```
"Hello, EdmontonPy!"
```

Decorating other people's code

Remember how I showed the "long way" of using decorators earlier, without the @-syntax? Well, that can be useful sometimes!

It can let you apply a decorator after a function is defined, and unlike the @-syntax, you can save the result with a new name:

```
foo = decorator(bar)
```

As a bonus, you can even do this to functions defined in other modules, without modifying their source code!

…well, except for methods on classes defined in the stdlib. So you can't redefine str.join (sadly - why must I write ''.join([string1, string2, string3]) when it could take a variable number of arguments?)

Also, you CAN save the result with the same name, but I think it would be local to your module, and you might be surprised what happens when code in other modules tries to use the function

Why and how decorators function

...Well, I would get into it if I had more time

But I will say this: If you thought **closures** were pointless and academic, think again!

- Not that the accepted answer for that question will help any, but the other answers are each a little bit helpful

If you want to know more, check out Matt Harrison's Guide To Python Decorators, a $5 ebook (or ask me!)

That's not an affiliate link, it's just a really good, concise explanation of all the things that are important behind the scenes. This includes functions as first class objects and closures, which seem tangential, but actually aren't.

Also, personal anecdote: Closures can turn ugly, awful code (like loops with multiple exit points that require cleanup) into really nice code. A++ would use again.

Useful decorators

Top three

- Deprecation that auto-calls the new function
- Turning recursive functions into loops (neat, but only useful if the function is called 2000+ times)
- Automatic caching (or for Python 3.2+, functools.lru_cach)

The rest (in no particular order)

- "Element-wise" functions
- Counting function calls, benchmarking (at the bottom)
- Abstract method decorator
- Goto decorator
- Python 3 type checking with annotations
- Immutable classes (can't be inherited from though)
- Deprecation which specifies the file and line of deprecated function
- Print the arguments of a function before calling it
- Synchronization for multi-threaded programs
- Redirecting a function's stdout
- Pre and post conditions
- Profiling and coverage analysis
- Timeout for long functions (POSIX systems only, sorry fellow Windows users :( )
- Variations of init methods that don't require any "self.x = x" junk: v1, v2, v3

Tools for writing decorators

Decorator module, as previously mentioned

The built-in functools.wraps

Venusian offers delayed decorator application, with the main goal of improving testability

Things I intentionally avoided here

Using classes for decorators, class decorators (they are different), some of the finer points of decorating methods (functions defined in a class)

See Bruce Eckel's Python 3 book for more detail on class related stuff, and some other links in my disorganized prep file