Blog :: Understanding Python Decorators in 12 Easy Steps!

1 July 2012

Ok, perhaps I jest. As a Python instructor, understanding decorators is a topic I find students consistently struggle with upon first exposure. That's because decorators are hard to understand! **Getting** decorators requires understanding several functional programming concepts as well as feeling comfortable with some unique features of Python's function definition and function calling syntax. *Using* decorators is easy (see Section 10)! But writing them can be complicated.

I can't make decorators easy - but maybe by walking through each piece of the puzzle one step at a time I can help you feel more confident in understanding decorators[1]. Because decorators are complex this is going to be a long article - but stick with it! I promise to make each piece as simple as possible - and if you understand each piece, you'll understand how decorators work! I'm trying to assume minimal Python knowledge but this will probably be most helpful to people who have at least a casual working exposure to Python.

I should also note that I used Python's doctest modules to run the Python code samples in this article. The code looks like an interactive Python console session (**>>>** and **...** indicate Python statements while output has its own line). There are occasionally cryptic comments that begin with "doctest" - these are just directives to doctest and can safely be ignored.

1. Functions

Functions in Python are created with the `def` keyword and take a name and an optional list of parameters. They can return values with the `return` keyword. Let's make and call the simplest possible function:

```
>>> def foo():
...     return 1
>>> foo()
1
```

The body of the function (as with all multi-line statements in Python) is mandatory and indicated by indentation. We can call functions by appending parentheses to the function name.

2. Scope

In Python functions create a new scope. Pythonistas might also say that functions have their own namespace. This means Python looks first in the namespace of the function to find variable names when it encounters them in the function body. Python includes a couple of functions that let us look at our namespaces. Let's write a simple function to investigate the difference between local and global scope.

```
>>> a_string = "This is a global variable"
>>> def foo():
...     print locals()
>>> print globals() # doctest: +ELLIPSIS
{..., 'a_string': 'This is a global variable'}
>>> foo() # 2
{}
```

The builtin `globals` function returns a dictionary containing all the variable names Python knows about. (For the sake of clarity I've omitted in the output a few variables Python automatically creates.) At point #2 I called my function `foo` which prints the contents of the local namespace inside the function. As we can see the function `foo` has its own separate namespace which is currently empty.

3. variable resolution rules

Of course this doesn't mean that we can't access global variables inside our function. Python's scope rule is that variable creation always creates a new local variable but variable access (including modification) looks in the local scope and then searches all the enclosing scopes to find a match. So if we modify our function `foo` to print our global variable things work as we would expect:

```
>>> a_string = "This is a global variable"
>>> def foo():
...     print a_string # 1
>>> foo()
This is a global variable
```

At point #1 Python looks for a local variable in our function and not finding one, looks for a global variable[2] of the same name.

On the other hand if we try to assign to the global variable inside our function it doesn't do what we want:

```
>>> a_string = "This is a global variable"
>>> def foo():
...     a_string = "test" # 1
...     print locals()
>>> foo()
{'a_string': 'test'}
>>> a_string # 2
'This is a global variable'
```

As we can see, global variables can be accessed (even changed if they are mutable data types) but not (by default) assigned to. At point #1 inside our function we are actually creating a new local variable that "shadows" the global variable with the same name. We can see this be by printing the `local` namespace inside our function `foo` and notice it now has an entry. We can also see back out in the global namespace at point #2 that when we check the value of the variable `a_string` it hasn't been changed at all.

4. Variable lifetime

It's also important to note that not only do variables live inside a namespace, they also have lifetimes. Consider

```
>>> def foo():
...     x = 1
>>> foo()
>>> print x # 1
Traceback (most recent call last):
  ...
NameError: name 'x' is not defined
```

It isn't just scope rules at point #1 that cause a problem (although that's why we have a NameError) it also has to do with how function calls are implemented in Python and many other languages. There isn't any syntax we can use to get the value of the variable `x` at this point - it literally doesn't exist! The namespace created for our function `foo` is created from scratch each time the function is called and it is destroyed when the function ends.

5. Function arguments and parameters

Python does allow us to pass arguments to functions. The parameter names become local variables in our function.

```
>>> def foo(x):
...     print locals()
>>> foo(1)
{'x': 1}
```

Python has a variety of ways to define function parameters and pass arguments to them. For the full skinny you'll want to see the Python documentation on defining functions. I'll give you the short version here: function parameters can be either **positional** parameters that are **mandatory** or **named, default value** parameters that are **optional**.

```
>>> def foo(x, y=0): # 1
...     return x - y
>>> foo(3, 1) # 2
2
>>> foo(3) # 3
3
>>> foo() # 4
Traceback (most recent call last):
  ...
TypeError: foo() takes at least 1 argument (0 given)
>>> foo(y=1, x=3) # 5
2
```

At point #1 we are defining a function that has a single positional parameter `x` and a single named parameter `y`. As we see at point #2 we can call this function passing arguments normally - the values are passed to the parameters of `foo` by position even though one is defined in the function definition as a named parameter. We can also call the function without passing any arguments at all for the named parameter as you can see at point #3 - Python uses the default value of 0 we declared if it doesn't receive a value for the named parameter `y`. Of course we can't leave out values for the first (mandatory, positional) parameter - point #4 demonstrates that this results in an exception.

All clear and straightforward? Now it gets slightly confusing - Python supports named arguments at function call time. Look at point #5 - here we are calling a function with two named arguments even though it was **defined** with one named and one positional parameter. Since we have names for our parameters the order we pass them in doesn't matter.

The opposite case is true of course. One of the parameters for our function is defined as a named parameter but we passed an argument to it by position - the call `foo(3,1)` at point #2 passes a 3 as the argument to our ordered parameter `x` and passes the second (an integer 1) to the second parameter even though it was defined as a named parameter.

Whoo! That feels like a lot of words to describe a pretty simple concept: function parameters can have names or positions. This means slightly different things depending on whether we're at function definition or function call time and we can use named arguments to functions defined only with positional parameters and vice-versa! Again - if that was all too rushed be sure to check out the the docs.

6. Nested functions

Python allows the creation of nested functions. This means we can declare functions inside of functions and all the scoping and lifetime rules still apply normally.

```
>>> def outer():
...     x = 1
...     def inner():
...         print x # 1
...     inner() # 2
```

```
...
>>> outer()
1
```

This looks a little more complicated, but it's still behaving in a pretty sensible manner. Consider what happens at point #1 - Python looks for a local variable named `x`, failing it then looks in the enclosing scope which is another function! The variable `x` is a local variable to our function `outer` but as before our function inner has access to the enclosing scope (read and modify access at least). At point #2 we call our inner function. It's important to remember that `inner` is also just a variable name that follows Python's variable lookup rules - Python looks in the scope of `outer` first and finds a local variable named `inner`.

7. Functions are first class objects in Python

This is simply the observation that in Python, functions are objects like everything else. Ah, function containing variable, you're not so special!

```
>>> issubclass(int, object) # all objects in Python inherit from a common baseclass
True
>>> def foo():
...     pass
>>> foo.__class__ # 1
<type 'function'>
>>> issubclass(foo.__class__, object)
True
```

You may never have thought of your functions as having attributes - but functions are objects in Python, just like everything else. (If you find that confusing wait till you hear that classes are objects in Python, just like everything else!) Perhaps this is making the point in an academic way - functions are just regular values like any other kind of value in Python. That means you can pass functions to functions as arguments or return functions from functions as return values! If you've never thought of this sort of thing consider the following perfectly legal Python:

```
>>> def add(x, y):
...     return x + y
>>> def sub(x, y):
...     return x - y
>>> def apply(func, x, y): # 1
...     return func(x, y) # 2
>>> apply(add, 2, 1) # 3
3
>>> apply(sub, 2, 1)
1
```

This example might not seem too strange too you - `add` and `sub` are normal Python functions that receive two values and return a calculated value. At point #1 you can see that the variable intended to receive a function is just a normal variable like any other. At point #2 we are calling the function passed into `apply` - parentheses in Python are the call operator and call the value the variable name contains. And at point #3 you can see that passing functions as values doesn't have any special syntax in Python - function names are just variable labels like any other variable.

You might have seen this sort of behavior before - Python uses functions as arguments for frequently used operations like customizing the `sorted` builtin by providing a function to the key parameter. But what about returning functions as values? Consider:

```
>>> def outer():
```

```
...     def inner():
...         print "Inside inner"
...     return inner # 1
...
>>> foo = outer() #2
>>> foo # doctest:+ELLIPSIS
<function inner at 0x...>
>>> foo()
Inside inner
```

This may seem a little more bizarre. At point #1 I return the variable `inner` which happens to be a function label. There's no special syntax here - our function is returning the inner function which otherwise couldn't be called. Remember variable lifetime? The function `inner` is freshly redefined each time the function `outer` is called but if `inner` wasn't returned from the function it would simply cease to exist when it went out of scope.

At point #2 we can catch the return value which is our function `inner` and store it in a new variable `foo`. We can see that if we evaluate `foo` it really does contain our function `inner` and we can call it by using the call operator (parentheses, remember?) This may look a little weird, but nothing to hard to understand so far, right? Hold on, because things are about to take a turn for the weird!

8. Closures

Let's not start with a definition, let's start with another code sample. What if we tweaked our last example slightly:

```
>>> def outer():
...     x = 1
...     def inner():
...         print x # 1
...     return inner
>>> foo = outer()
>>> foo.func_closure # doctest: +ELLIPSIS
(<cell at 0x...: int object at 0x...>,)
```

From our last example we can see that `inner` is a function returned by `outer`, stored in a variable named `foo` and we could call it with `foo()`. But will it run? Let's consider scoping rules first.

Everything works according to Python's scoping rules - `x` is a local variable in our function `outer`. When `inner` prints `x` at point #1 Python looks for a local variable to `inner` and not finding it looks in the enclosing scope which is the function `outer`, finding it there.

But what about things from the point of view of variable lifetime? Our variable `x` is local to the function `outer` which means it only exists while the function `outer` is running. We aren't able to call `inner` till after the return of `outer` so according to our model of how Python works, `x` shouldn't exist anymore by the time we call `inner` and perhaps a runtime error of some kind should occur.

It turns out that, against our expectations, our returned `inner` function does work. Python supports a feature called **function closures** which means that inner functions defined in non-global scope remember what their enclosing namespaces looked like **at definition time**. This can be seen by looking at the `func_closure` attribute of our inner function which contains the variables in the enclosing scopes.

Remember - the function inner is being newly defined each time the function outer is called. Right now the value of `x` doesn't change so each `inner` function we get back does the same thing as another `inner` function - but what if we tweaked it a little bit?

```
>>> def outer(x):
...     def inner():
...         print x # 1
...     return inner
>>> print1 = outer(1)
>>> print2 = outer(2)
>>> print1()
1
>>> print2()
2
```

From this example you can see that **closures** - the fact that functions remember their enclosing scope - can be used to build custom functions that have, essentially, a hard coded argument. We aren't passing the numbers 1 or 2 to our `inner` function but are building custom versions of our inner function that "remembers" what number it should print.

This alone is a powerful technique - you might even think of it as similar to object oriented techniques in some ways: `outer` is a constructor for `inner` with `x` acting like a private member variable. And the uses are numerous - if you are familiar with the `key` parameter in Python's `sorted` function you have probably written a lambda function to sort a list of lists by the second item instead of the first. You might now be able to write an `itemgetter` function that accepts the index to retrieve and returns a function that could suitably be passed to the key parameter.

But let's not do anything so mundane with closures! Instead let's stretch one more time and write a decorator!

9. Decorators!

A decorator is just a callable that takes a function as an argument and returns a replacement function. We'll start simply and work our way up to useful decorators.

```
>>> def outer(some_func):
...     def inner():
...         print "before some_func"
...         ret = some_func() # 1
...         return ret + 1
...     return inner
>>> def foo():
...     return 1
>>> decorated = outer(foo) # 2
>>> decorated()
before some_func
2
```

Look carefully through our decorator example. We defined a function named `outer` that has a single parameter `some_func`. Inside `outer` we define an nested function named `inner`. The `inner` function will print a string then call `some_func`, catching its return value at point #1. The value of `some_func` might be different each time `outer` is called, but whatever function it is we'll call it. Finally `inner` returns the return value of `some_func()` + 1 - and we can see that when we call our returned function stored in `decorated` at point #2 we get the results of the print and also a return value of 2 instead of the original return value 1 we would expect to get by calling `foo`.

We could say that the variable `decorated` is a decorated version of `foo` - it's `foo` plus something. In fact if we wrote a useful decorator we might want to replace `foo` with the decorated version altogether so we always got our "plus something" version of foo. We can do that without learning any new syntax simply

by re-assigning the variable that contains our function:

```
>>> foo = outer(foo)
>>> foo # doctest: +ELLIPSIS
<function inner at 0x...>
```

Now any calls to `foo()` won't get the original `foo`, they'll get our decorated version! Got the idea? Let's write a more useful decorator.

Imagine we have a library that gives us coordinate objects. Perhaps they are primarily made up of `x` and `y` coordinate pairs. Sadly the coordinate objects don't support mathematical operators and we can't modify the source so we can't add this support ourselves. We're going to be doing a bunch of math, however, so we want to make `add` and `sub` functions that take two coordinate objects and do the appropriate mathematical thing. These functions would be easy to write (I'll provide a sample Coordinate class for the sake of illustration)

```
>>> class Coordinate(object):
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...     def __repr__(self):
...         return "Coord: " + str(self.__dict__)
>>> def add(a, b):
...     return Coordinate(a.x + b.x, a.y + b.y)
>>> def sub(a, b):
...     return Coordinate(a.x - b.x, a.y - b.y)
>>> one = Coordinate(100, 200)
>>> two = Coordinate(300, 200)
>>> add(one, two)
Coord: {'y': 400, 'x': 400}
```

But what if our add and subtract functions had to also have some bounds checking behavior? Perhaps you can only sum or subtract based on positive coordinates and any result should be limited to positive coordinates as well. So currently

```
>>> one = Coordinate(100, 200)
>>> two = Coordinate(300, 200)
>>> three = Coordinate(-100, -100)
>>> sub(one, two)
Coord: {'y': 0, 'x': -200}
>>> add(one, three)
Coord: {'y': 100, 'x': 0}
```

but we'd rather have have the difference of `one` and `two` be $\{x: 0, y: 0\}$ and the sum of `one` and `three` be $\{x: 100, y: 200\}$ without modifying `one`, `two`, or `three`. Instead of adding bounds checking to the input arguments of each function and the return value of each function let's write a bounds checking decorator!

```
>>> def wrapper(func):
...     def checker(a, b): # 1
...         if a.x < 0 or a.y < 0:
...             a = Coordinate(a.x if a.x > 0 else 0, a.y if a.y > 0 else 0)
...         if b.x < 0 or b.y < 0:
...             b = Coordinate(b.x if b.x > 0 else 0, b.y if b.y > 0 else 0)
...         ret = func(a, b)
```

```
...         if ret.x < 0 or ret.y < 0:
...             ret = Coordinate(ret.x if ret.x > 0 else 0, ret.y if ret.y > 0 else 0)
...         return ret
...     return checker
>>> add = wrapper(add)
>>> sub = wrapper(sub)
>>> sub(one, two)
Coord: {'y': 0, 'x': 0}
>>> add(one, three)
Coord: {'y': 200, 'x': 100}
```

This decorator works just as before - returns a modified version of a function but in this case it does something useful by checking and normalizing the input parameters and the return value, substituting 0 for any negative x or y values.

It's a matter of opinion as to whether doing it this makes our code cleaner: isolating the bounds checking in its own function and applying it to all the functions we care to by wrapping them with a decorator. The alternative would be a function call on each input argument and on the resulting output before returning inside each math function and it is undeniable that using the decorator is at least less repetitious in terms of the amount of code needed to apply bounds checking to a function. In fact - if its our own functions we're decorating we could make the decorator application a little more obvious.

10. The `@` symbol applies a decorator to a function

Python 2.4 provided support to wrap a function in a decorator by pre-pending the function definition with a decorator name and the @ symbol. In the code samples above we decorated our function by replacing the variable containing the function with a wrapped version.

```
>>> add = wrapper(add)
```

This pattern can be used at any time, to wrap any function. But if we are defining a function we can "decorate" it with the @ symbol like:

```
>>> @wrapper
... def add(a, b):
...     return Coordinate(a.x + b.x, a.y + b.y)
```

It's important to recognize that this is no different than simply replacing the original variable `add` with the return from the `wrapper` function - Python just adds some syntactic sugar to make what is going on very explicit.

Again - using decorators is easy! Even if writing useful decorators like `staticmethod` or `classmethod` would be difficult, using them is just a matter of prepending your function with `@decoratorname`!

11. *args and **kwargs

We've written a useful decorator but it's hard coded to work only on a particular kind of function - one which takes two arguments. Our inner function `checker` accepts two arguments and passes the arguments on to the function captured in the closure. What if we wanted a decorator that did something for any possible function? Let's write a decorator that increments a counter for every function call of every decorated function without changing any of it's decorated functions. This means it would have to accept the calling signature of any of the functions that it decorates and also call the functions it decorates passing on whatever arguments were passed to it.

It just so happens that Python has syntactic support for just this feature. Be sure to read the Python Tutorial for more details but the `*` operator used when defining a function means that any extra positional arguments passed to the function end up in the variable prefaced with a `*`. So:

```
>>> def one(*args):
...     print args # 1
>>> one()
()
>>> one(1, 2, 3)
(1, 2, 3)
>>> def two(x, y, *args): # 2
...     print x, y, args
>>> two('a', 'b', 'c')
a b ('c',)
```

The first function `one` simply prints whatever (if any) positional arguments are passed to it. As you can see at point #1 we simply refer to the variable `args` inside the function - `*args` is only used in the function definition to indicate that positional arguments should be stored in the variable `args`. Python also allows us to specify some variables and catch any additional parameters in `args` as we can see at point #2.

The `*` operator can also be used when calling functions and here it means the analogous thing. A variable prefaced by `*` when **calling** a function means that the variable contents should be extracted and used as positional arguments. Again by example:

```
>>> def add(x, y):
...     return x + y
>>> lst = [1,2]
>>> add(lst[0], lst[1]) # 1
3
>>> add(*lst) # 2
3
```

The code at point #1 does exactly the same thing as the code at point #2 - Python is doing automatically for us at point #2 what we could do manually for ourselves. This isn't too bad - `*args` means either extract positional variables from an iterable if calling a function or when defining a function accept any extra positional variables.

Things get only slightly more complicated when we introduce `**` which does for dictionaries & key/value pairs exactly what `*` does for iterables and positional parameters. Simple, right?

```
>>> def foo(**kwargs):
...     print kwargs
>>> foo()
{}
>>> foo(x=1, y=2)
{'y': 2, 'x': 1}
```

When we define a function we can use `**kwargs` to indicate that all uncaptured keyword arguments should be stored in a dictionary called `kwargs`. As before neither the name `args` nor `kwargs` is part of Python syntax but it is convention to use these variable names when declaring functions. Just like `*` we can use `**` when calling a function as well as when defining it.

```
>>> dct = {'x': 1, 'y': 2}
>>> def bar(x, y):
...     return x + y
>>> bar(**dct)
3
```

12. More generic decorators

Given our new power we can write a decorator that "logs" the arguements to functions. We'll just print to stdout for simplicity sake:

```
>>> def logger(func):
...     def inner(*args, **kwargs): #1
...         print "Arguments were: %s, %s" % (args, kwargs)
...         return func(*args, **kwargs) #2
...     return inner
```

Notice our `inner` function takes any arbitrary number and type of parameters at point #1 and passes them along as arguments to the wrapped function at point #2. This allows us to wrap or decorate any function, no matter it's signature.

```
>>> @logger
... def foo1(x, y=1):
...     return x * y
>>> @logger
... def foo2():
...     return 2
>>> foo1(5, 4)
Arguments were: (5, 4), {}
20
>>> foo1(1)
Arguments were: (1,), {}
1
>>> foo2()
Arguments were: (), {}
2
```

Calling our functions results in a "logging" output line as well as the expected return value of each function.

More about decorators

If you followed the last example you understand decorators! Congratulations - Go forth and use your new powers for good!

You might also consider a little further study: Bruce Eckel has an excellent essay on decorators and implements them in Python with objects instead of functions. You might find the OOP code easier to read than our purely functional version. Bruce also has a follow-up essay on providing arguments to decorators that may also easier to implement with objects than with functions. Finally - you might also investigate the builtin functools `wraps` function which (confusingly) is a decorator that can be used in our decorators to modify the signature of our replacement functions so they look more like the decorated function.

[1] I also recently read an essay on explaining decorators that set me thinking…

[2] "global" is a big fat lie in Python which is a wonderful thing, but a discussion for another time…

Update: Thanks to Nick I've updated my terminology throughout to be clear that "parameters" are the named variables in function signatures while "arguments" are the values passed to functions.

Update: I've fixed several typos in the article. Thanks Gregory!
