Python decorators

Reuven M. Lerner, PhD
reuven@lerner.co.il

Decorator background

- Remember:
  - Functions can be passed as function parameters
  - Functions can be returned from functions
- Anything that we can execute is known as "callable" in Python. Normally, these are functions and classes.

Example: Static methods

```
@staticmethod

def sthing():

    print "Hi  from static method."
```

Example: Static methods

- This is precisely the same as saying:

```
def sthing():
    print "Hi from static method."
sthing = staticmethod(sthing)
```

- Notice that we have redefined "sthing" as the result of invoking "staticmethod" on the original "sthing"!

@ makes it a decorator

- @ tells Python that the function (or class) we are about to define shouldn't be assigned to the name in the "def" (or "class") statement.
- Rather, we should create the new function (or class) object, and then pass it to the decorator!
- The result of invoking the decorator is then assigned to the def/class name.

So…

- If I say

```
@foo
def bar():
    return 5
```

- The value of "bar" is the result of invoking foo(bar)
- The original "bar" function lacks a name

What is a decorator?

- @staticmethod is a “decorator”, thanks to the @
- Decorators are sort of like macros, or Lisp "advice"
- Decorators are callables (normally a function, but it can be a class that implements __call__)
  - Receives the compiled function, and can then do with it what it likes
- Examples: @staticmethod, @classmethod, @property

Decorator options

- Class decorator on a function
- Function decorator on a function
- Class decorator on a class
- Class decorator on a function

That's a lot of options!

- We'll first talk about writing the decorator as a class, and using it to decorate functions — but all of these options are possible.
- If you want to define your decorator as a function, you will need nested functions!

Defining a decorator class

- A new instance of the class is created each time the decorator is applied
- The decorator application happens when a function is defined
- That is, __init__ will be invoked when the new function is defined!
- __init__ won't get parameter

Function invocation

- Invocation of a callable is done with __call__
- Your __call__ method will get all of the parameters that would normally be passed to the function
- You can then call the function with (), with or without parameters (as you see fit)

Decorator class summary

- Define a class
- __init__ will handle function definition; the function is passed as the second parameter (after self)
- __call__ will handle function invocation
- If you want, set such things as __doc__ and __name__ to make the wrapped function behave appropriately

Writing a decorator class

```
class myDecorator(object):


    def __init__(self, f):
        print "inside myDecorator.__init__()"
        self.f = f


    def __call__(self):
        print "about to call the function"
        self.f()
        print "just called the function"
```

Using the decorator class

```
@myDecorator

def foo():

  print "Hello from inside of foo"

inside myDecorator.__init__()

>>> foo()

about to call the function

Hello from inside of foo

just called the function
```

To summarize

- A new instance of our decorator is created at function-definition time
- The function is put in an attribute on that instance, in __init__
- When we invoke the function, we're really invoking __call__ on the instance … which then invokes our original function

Who needs this?

- Logging
- Type checking
- Sanity checking
- Environment checking (e.g., authorization)
- Starting/stopping events, such as database transactions
- Benchmarking
- Share code across functions, classes

Decorator ideas

- For lots of ideas involving decorators (many of which use the function syntax), check out:
- https://wiki.python.org/moin/PythonDecoratorLibrary