

Python Decorators

Stéphane Vialette

LIGM, Université Paris-Est Marne-la-Vallée

November 14, 2011

Outline

- 1 Introduction
- 2 Decorators without arguments
- 3 Decorators with arguments

Outline

- 1 Introduction
- 2 Decorators without arguments
- 3 Decorators with arguments

Decorators

Description

- Decorators are just overlays on top of function calls. These overlays are just additional calls that are applied when a function or method is declared.
- The syntax for decorators uses a leading *at-sign* (@) followed by the decorator name and optional arguments.

Syntaxe

```
func = decorator1(decorator2(decorator3(func(arg1, arg2, ...))))
```

Stacking

Description

Stacking multiple decorators

```
@decorator1
@decorator2
@decorator3
def func(arg1, arg2, ...):
    pass
```

is equivalent to creating a composite function

```
func = decorator1(decorator2(decorator3(func(arg1, arg2, ...))))
```

Outline

- 1 Introduction
- 2 Decorators without arguments
- 3 Decorators with arguments

Decorators without arguments

Example

```
def entry_exit(f):
    def wrapped():
        print ">> Entering", f.__name__, '>>' ,
        f()
        print "<< Exited", f.__name__, '<<'
    return wrapped

@entry_exit
def func():
    print "Inside func",

@entry_exit
def func2():
    print "inside func2",

if __name__ == '__main__':
    func()  # >> Entering func >> Inside func << Exited func <<
    func2() # >> Entering func2 >> inside func2 << Exited func2 <<
```

Measuring the running time

Example

```
import time

def timeit(func):
    def timed(*args, **kw):
        ts = time.time()
        result = func(*args, **kw)
        te = time.time()
        print '%r (%r, %r) %2.2f sec' % \
            (func.__name__, args, kw, te-ts)
    return result
return timed
```

Measuring the running time

Example

```
import time
from timeit import timeit

class C(object):
    @timeit
    def f(self):
        time.sleep(0.2)

@timeit
def g():
    time.sleep(1)

c = C()
c.f() # 'f' ((<__main__.C object at 0xb7e2df4c>,), {}) 0.20 sec
g() # 'g' ((), {}) 1.00 sec
```

Check that the supplied argument is an integer

Example

```
def requireint(func):  
    def wrapper (arg):  
        assert isinstance(arg, int)  
        return func(arg)  
    return wrapper  
  
@require_int  
def g(arg):  
    ...
```

Decorators as classes

Description

- The only constraint upon the object returned by the decorator is that it can be used as a function – which basically means it must be callable.
- Thus, any classes we use as decorators must implement `__call__`.

```
class MyDecorator(object):  
    def __init__(self, func):  
        # do something with func ...  
    def __call__(*args):  
        # do something ...
```

Classes as decorators

Example

```
In [1]: class Decorator(object):
...:     def __init__(self, func):
...:         print 'Decorator.__init__'
...:         self.__func = func
...:     def __call__(self, *args):
...:         print 'Decorator.__call__'
...:         return self.__func(args)

In [2]: @Decorator
...: def f(arg):
...:     print 'f(%s)' % (arg,)
Decorator.__init__
In [3]: Decorator
Out[3]: <class '.__main__.Decorator'>
In [4]: f
Out[4]: <.__main__.Decorator object at 0x913f12c>
In [5]: f('arg')
Decorator.__call__
f(('arg',))
```

Counting function calls

Example

```
class countcalls(object):  
  
    def __init__(self, func):  
        self.__func = func  
        self.__numcalls = 0  
  
    def __call__(self, *args, **kwargs):  
        self.__numcalls += 1  
        return self.__func(*args, **kwargs)  
  
    def count(self):  
        return self.__numcalls
```

Counting function calls

Example

```
@countcalls
```

```
def f1():  
    pass
```

```
@countcalls
```

```
def f2():  
    pass
```

```
print 'f1:', f1
```

```
print 'f2:', f2
```

```
for _ in range(3):  
    f1()
```

```
for _ in range(2):  
    f1()  
    f2()
```

```
print f1.count() # would print 5
```

```
print f2.count() # would print 2
```

Counting function calls

This does not work !

```
class countcalls(object):
    instances = {}

    def __init__(self, func):
        self.func = func
        self.numcalls = 0
        countcalls.instances[func] = self

    def __call__(self, *args, **kwargs):
        self.numcalls += 1
        return self.func(*args, **kwargs)

    @staticmethod
    def count(func):
        return countcalls.instances[func].numcalls

    @staticmethod
    def counts():
        return dict([(func, countcalls.count(func)) for func in countcalls.instances])
```

Counting function calls

This does not work !

```
In [1]: from countcalls import countcalls
```

```
In [2]: @countcalls
...: def f():
...:     pass
...:
```

```
In [3]: f()
```

```
In [4]: print countcalls.count(f)
```

```
-----
KeyError Traceback (most recent call last)
.../decorator/src/<ipython console> in <module>()
.../decorator/src/badcountcalls.pyc in count(func)
    13     @staticmethod
    14     def count(func):
----> 15         return countcalls.instances[func].numcalls
    16
    17     @staticmethod
```

```
KeyError: <badcountcalls.countcalls object at 0x9b422ac>
```

Counting function calls

This does not work !

```
class countcalls(object):
    instances = {}

    def __init__(self, func):
        self.func = func
        print "I am decorating ", self.func
        self.numcalls = 0
        countcalls.instances[func] = self

    def __call__(self, *args, **kwargs):
        self.numcalls += 1
        return self.func(*args, **kwargs)

    @staticmethod
    def count(func):
        return countcalls.instances[func].numcalls

    @staticmethod
    def counts():
        return dict([(func, countcalls.count(func)) for func in countcalls.instances])
```

Counting function calls

This does not work !

```
In [1]: from countcalls import countcalls
```

```
In [2]: @countcalls
```

```
...: def f():
```

```
...:     pass
```

```
...:
```

```
I am decorating <function f at 0x93353e4>
```

```
In [3]: print f
```

```
-----> print(f)
```

```
<badcountcalls3.countcalls object at 0x93402ac>
```

```
In [4]: # Et voila ... !
```

Generating deprecation warnings

Example

```
import warnings

def deprecated(func):
    """This is a decorator which can be used to mark functions
    as deprecated. It will result in a warning being emitted
    when the function is used."""
    def new_func(*args, **kwargs):
        warnings.warn("Call to deprecated function %s." % func.__name__,
                      category=DeprecationWarning)
        return func(*args, **kwargs)
    new_func.__name__ = func.__name__
    new_func.__doc__ = func.__doc__
    new_func.__dict__.update(func.__dict__)
    return new_func
```

Generating deprecation warnings

Example

```
In [1]: from deprecation import deprecated
In [2]: @deprecated
...: def some_old_fun():
...:     pass
...:
In [3]: some_old_fun()
deprecation.py:9: DeprecationWarning: Call to deprecated function some_old_fun.
category=DeprecationWarning)
In [4]: class A(object):
...:     @deprecated
...:     def some_old_method(self):
...:         pass
...:
In [5]: a = A()
In [6]: a.some_old_method()
deprecation.py:9: DeprecationWarning: Call to deprecated function some_old_method.
category=DeprecationWarning)
```

Pseudo-carrying

Example

```
class curried(object):
    """
    Decorator that returns a function that keeps returning functions
    until all arguments are supplied; then the original function is
    evaluated.
    """
    def __init__(self, func, *args):
        self.func = func
        self.args = args

    def __call__(self, *args):
        args = self.args + args
        if len(args) < self.func.func_code.co_argcount:
            return curried(self.func, *args)
        else:
            return self.func(*args)
```

Pseudo-carrying

Testing

```
In [1]: from curry import curried
```

```
In [2]: @curried
...: def add(x, y):
...:     return x+y
...:
```

```
In [3]: add1 = add(1)
```

```
In [4]: print add1
<pcarrying.curried object at 0x8e96c0c>
```

```
In [5]: add1(2)
Out[5]: 3
```

```
In [6]: add2 = add(2)
```

```
In [7]: add2(3)
Out[7]: 5
```

Outline

- 1 Introduction
- 2 Decorators without arguments
- 3 Decorators with arguments

Decorators with arguments

Description

Decorators can have arguments, like in

```
@decorator(arg)
def func():
    ...
```

is equivalent to creating a composite function

```
func = decorator(arg)(func) # decorator(arg) must be a decorator
```

Adding an attribute

Example

```
In [1]: def option(value):
...:     def decorator(func):
...:         func.attribute = value
...:         return func
...:     return decorator
...:
In [2]: @option('a value')
...: def f():
...:     pass
...:
In [3]: f()
In [4]: f.attribute
Out[4]: 'a value'
```

Checking type

Example

```
def return_bool(bool_value):  
    def wrapper(func):  
        def wrapped(*args):  
            result = func(*args)  
            if result != bool:  
                raise TypeError  
            return result  
        return wrapped  
    return wrapper  
  
@return_bool(True)  
def always_true():  
    return True  
always_true() # ok  
  
@return_bool(False)  
def always_false():  
    return True  
always_false() # raise TypeError
```

Easy adding methods to a class instance 1/3

Example

```
In [1]: import new
```

```
In [2]: newinstancemethod?
```

```
Type:                type
```

```
Base Class:          <type 'type'>
```

```
String Form:         <type 'instancemethod'>
```

```
Namespace:           Interactive
```

```
Docstring:
```

```
    instancemethod(function, instance, class)
```

```
    Create an instance method object.
```

```
In [3]:
```

Easy adding methods to a class instance 2/3

Example

```
# Add a new method to a class instance.  
# Credits to John Roth.  
def addto(instance):  
    def decorator(func):  
        import new  
        func = new.instancemethod(func,  
                                   instance,  
                                   instance.__class__)  
        setattr(instance, func.func_name, func)  
        return func  
    return decorator
```

Easy adding methods to a class instance 3/3

Example

```
class C(object):  
    def __init__(self, val):  
        self.attribute = val
```

```
c1 = C(123)  
c2 = C(456)
```

```
@addto(c1)  
def print_attribute(self):  
    print self.attribute
```

```
c1.print_attribute() # would print 123
```

```
c2.print_attribute() # AttributeError: 'C' object has no attribute
```