Decorators

Python 2.x 2.4–2.7 , Python 3.x 3.0–3.6

Decorator functions are software design patterns. They dynamically alter the functionality of a function, method, or class without having to directly use subclasses or change the source code of the decorated function. When used correctly, decorators can become powerful tools in the development process. This topic covers implementation and applications of decorator functions in Python.

Examples

Improvements requested:

Decorator function

Decorators augment the behavior of other functions or methods. Any function that takes a function as a parameter and returns an augmented function can be used as a **decorator** .

```
# This simplest decorator does nothing to the function being decorated. Such
# minimal decorators can occasionally be used as a kind of code markers.
def super_secret_function(f):
    return f

@super_secret_function
def my_function():
    print("This is my secret function.")
```

The `@` -notation is syntactic sugar that is equivalent to the following:

```
my_function = super_secret_function(my_function)
```

It is important to bear this in mind in order to understand how the decorators work. This "unsugared" syntax makes it clear why the decorator function takes a function as an argument, and why it should return another function. It also demonstrates what would happen if you *don't* return a function:

```
def disabled(f):
    """
    This function returns nothing, and hence removes the decorated function
    from the local scope.
    """
    pass

@disabled
def my_function():
    print("This function can no longer be called...")

my_function()
# TypeError: 'NoneType' object is not callable
```

Thus, we usually define a *new function* inside the decorator and return it. This new function would first do something that it needs to do, then call the original function, and finally process the return value. Consider the following example of a simple logger decorator:

```
def log_function_calls(func):
    """
    Augment the target function, so that whenever someone calls it we will
    log a message to the stdout.
    """
    def wrapped_func(*args, **kwargs):
        # Print a log message before the function is called.
        str_args = ", ".join(repr(a) for a in args)
        str_kwargs = ", ".join("%s=%r" % (k, v) for k, v in kwargs.items())
        str_all = ", ".join([str_args, str_kwargs])
        print("Calling %s(%s)..." % (func.__name__, str_all))
        try:
            # Call the original function.
            res = func(*args, **kwargs)
            # Log the returned result.
            print("%s(...) returned: %r" % (func.__name__, res))
            # Return the original result.
            return res
        except Exception as e:
            # Log the fact that the function threw an exception.
            print("Raised an exception %s" % e)
            # Re-raise the original exception.
            raise
    # Return the wrapped_func, which will replace the function being decorated.
    return wrapped_func

@log_function_calls
def testfunc(*args, **kwargs):
    print("  This is my function I'd like to debug")
    print("  This is my function I'd like to debug")
    print("  Got %d *args and %d **kwargs" % (len(args), len(kwargs)))
    return "".join(args) * kwargs.get("count", 1)

# Try it out:
hoho = testfunc("h", "o", count=10)
print()
apples = testfunc("apple", count="orange")
```

Output:

```
Calling testfunc('h', 'o', count=10)
  This is my function I'd like to debug
  Got 2 *args and 1 **kwargs
testfunc(...) returned: 'hohohohohohohohohoho'

Calling testfunc('apple', count='orange')
  This is my function I'd like to debug
  Got 1 *args and 1 **kwargs
Raised an exception can't multiply sequence by non-int of type 'str'
TypeError: can't multiply sequence by non-int of type 'str'
```

Decorator class

As mentioned in the introduction, a decorator is a function that can be applied to another function to augment its behavior. The syntactic sugar is equivalent to the following: `my_func = decorator(my_func)` . But what if the `decorator` was instead a class? The syntax would still work, except that now `my_func` gets replaced with an instance of the `decorator` class. If this class implements the `__call__()` magic method, then it would still be possible to use `my_func` as if it was a function:

```
class Decorator(object):
    """Simple decorator class."""

    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print('Before the function call.')
        res = self.func(*args, **kwargs)
        print('After the function call.')
        return res

@Decorator
def testfunc():
    print('Inside the function.')

testfunc()
# Before the function call.
# Inside the function.
# After the function call.
```

Note that a function decorated with a class decorator will no longer be considered a "function" from type-checking perspective:

```
import types
isinstance(testfunc, types.FunctionType)
# False
type(testfunc)
# <class '__main__.Decorator'>
```

Decorating Methods

For decorating methods you need to define an additional `__get__` -method:

```
from types import MethodType

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

    def __call__(self, *args, **kwargs):
        print('Inside the decorator.')
        return self.func(*args, **kwargs)

    def __get__(self, instance, cls):
        # Return a Method if it is called on an instance
        return self if instance is None else MethodType(self, instance)

class Test(object):
    @Decorator
    def __init__(self):
        pass

a = Test()
```

```
Inside the decorator.
```

Warning!

Class Decorators only produce one instance for a specific function so decorating a method with a class decorator will share the same decorator between all instances of that class:

```
from types import MethodType

class CountCallsDecorator(object):
    def __init__(self, func):
        self.func = func
        self.ncalls = 0    # Number of calls of this method

    def __call__(self, *args, **kwargs):
        self.ncalls += 1   # Increment the calls counter
        return self.func(*args, **kwargs)

    def __get__(self, instance, cls):
        return self if instance is None else MethodType(self, instance)

class Test(object):
    def __init__(self):
        pass

    @CountCallsDecorator
    def do_something(self):
        return 'something was done'

a = Test()
a.do_something()
a.do_something.ncalls   # 1
b = Test()
b.do_something()
b.do_something.ncalls   # 2
```

Decorator with arguments (decorator factory)

A decorator takes just one argument: the function to be decorated. There is no way to pass other arguments.

But additional arguments are often desired. The trick is then to make a function which takes arbitrary arguments and returns a decorator.

Decorator functions

```
def decoratorfactory(message):
    def decorator(func):
        def wrapped_func(*args, **kwargs):
            print('The decorator wants to tell you: {}'.format(message))
            return func(*args, **kwargs)
        return wrapped_func
    return decorator

@decoratorfactory('Hello World')
def test():
    pass

test()
```

```
The decorator wants to tell you: Hello World
```

Important Note:

With such decorator factories you **must** call the decorator with a pair of parentheses:

```
@decoratorfactory # Without parentheses
def test():
    pass

test()
```

```
TypeError: decorator() missing 1 required positional argument: 'func'
```

Decorator classes

```
def decoratorfactory(*decorator_args, **decorator_kwargs):

    class Decorator(object):
```

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

        def __call__(self, *args, **kwargs):
            print('Inside the decorator with arguments {}'.format(decorator_args))
            return self.func(*args, **kwargs)

    return Decorator

@decoratorfactory(10)
def test():
    pass

test()
```

```
Inside the decorator with arguments (10,)
```

Making a decorator look like the decorated function

Decorators normally strip function metadata as they aren't the same. This can cause problems when using meta-programming to dynamically access function metadata. Metadata also includes function's docstrings and its name. functools.wraps makes the decorated function look like the original function by copying several attributes to the wrapper function.

```
from functools import wraps
```

The two methods of wrapping a decorator are achieving the same thing in hiding that the original function has been decorated. There is no reason to prefer the function version to the class version unless you're already using one over the other.

As a function

```
def decorator(func):
    # Copies the docstring, name, annotations and module to the decorator
    @wraps(func)
    def wrapped_func(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapped_func

@decorator
def test():
    pass

test.__name__
```

```
'test'
```

As a class

```
class Decorator(object):
    def __init__(self, func):
        # Copies name, module, annotations and docstring to the instance.
        self._wrapped = wraps(func)(self)

    def __call__(self, *args, **kwargs):
        return self._wrapped(*args, **kwargs)

@Decorator
def test():
    """Docstring of test."""
    pass

test.__doc__
```

```
'Docstring of test.'
```

Create singleton class with a decorator

A singleton is a pattern that restricts the instantiation of a class to one instance/object. Using a decorator, we can define a class as a singleton by forcing the class to either return an existing instance of the class or create a new instance (if it doesn't exist).

```
def singleton(cls):
    instance = [None]
    def wrapper(*args, **kwargs):
        if instance[0] is None:
            instance[0] = cls(*args, **kwargs)
        return instance[0]

    return wrapper
```

This decorator can be added to any class declaration and will make sure that at most one instance of the class is created. Any subsequent calls will return the already existing class instance.

```
@singleton
class SomeSingletonClass:
    x = 2
    def __init__(self):
        print("Created!")

instance = SomeSingletonClass()  # prints: Created!
instance = SomeSingletonClass()  # doesn't print anything
print(instance.x)                # 2

instance.x = 3
print(SomeSingletonClass().x)    # 3
```

So it doesn't matter whether you refer to the class instance via your local variable or whether you create another "instance", you always get the same object.

Using a decorator to time a function

```
import time
def timer(func):
    def inner(*args, **kwargs):
        t1 = time.time()
        f = func(*args, **kwargs)
        t2 = time.time()
        print 'Runtime took {0} seconds'.format(t2-t1)
        return f
    return inner

@timer
def example_function():
    #do stuff
```

Syntax

```
def decorator_function(f): pass # defines a decorator named decorator_function
```

```
@decorator_function
def decorated_function(): pass # the function is now wrapped (decorated by)
decorator_function
```

```
decorated_function = decorator_function(decorated_function) # this is equivalent to using
the syntactic sugar @decorator_function
```

Parameters

Parameter	Details
f	The function to be decorated (wrapped)

Remarks