The decorator module

Author:	Michele Simionato
E-mail:	michele.simionato@gmail.com

Version:	3.0.1 (2009-02-16)
Requires:	Python 2.4+
Download page:	http://pypi.python.org/pypi/decorator/3.0.1
Installation:	easy_install decorator
License:	BSD license

Contents

Introduction

Python decorators are an interesting example of why syntactic sugar matters. In principle, their introduction in Python 2.4 changed nothing, since they do not provide any new functionality which was not already present in the language. In practice, their introduction has significantly changed the way we structure our programs in Python. I believe the change is for the best, and that decorators are a great idea since:

- decorators help reducing boilerplate code;
- decorators help separation of concerns;
- decorators enhance readability and maintenability;
- decorators are explicit.

Still, as of now, writing custom decorators correctly requires some experience and it is not as easy as it could be. For instance, typical implementations of decorators involve nested functions, and we all know that flat is better than nested.

The aim of the `decorator` module it to simplify the usage of decorators for the average programmer, and to popularize decorators by showing various non-trivial examples. Of course, as all techniques, decorators can be abused (I have seen that) and you should not try to solve every problem with a decorator, just because you can.

You may find the source code for all the examples discussed here in the `documentation.py` file, which contains this documentation in the form of doctests.

Definitions

Technically speaking, any Python object which can be called with one argument can be used as a decorator. However, this definition is somewhat too large to be really useful. It is more convenient to split the generic class of decorators in two subclasses:

- *signature-preserving* decorators, i.e. callable objects taking a function as input and returning a function *with the same signature* as output;
- *signature-changing* decorators, i.e. decorators that change the signature of their input function, or decorators returning non-callable objects.

Signature-changing decorators have their use: for instance the builtin classes `staticmethod` and `classmethod` are in this group, since they take functions and return descriptor objects which are not functions, nor callables.

However, signature-preserving decorators are more common and easier to reason about; in particular signature-preserving decorators can be composed together whereas other decorators in general cannot.

Writing signature-preserving decorators from scratch is not that obvious, especially if one wants to define proper decorators that can accept functions with any signature. A simple example will clarify the issue.

Statement of the problem

A very common use case for decorators is the memoization of functions. A `memoize` decorator works by caching the result of the function call in a dictionary, so that the next time the function is called with the same input parameters the result is retrieved from the cache and not recomputed. There are many implementations of `memoize` in http://www.python.org/moin/PythonDecoratorLibrary, but they do not preserve the signature. A simple implementation for Python 2.5 could be the following (notice that in general it is impossible to memoize correctly something that depends on non-hashable arguments):

```
def memoize25(func):
    func.cache = {}
    def memoize(*args, **kw):
        if kw: # frozenset is used to ensure hashability
            key = args, frozenset(kw.iteritems())
        else:
            key = args
        cache = func.cache
        if key in cache:
            return cache[key]
        else:
            cache[key] = result = func(*args, **kw)
            return result
    return functools.update_wrapper(memoize, func)
```

Here we used the functools.update_wrapper utility, which has been added in Python 2.5 expressly to simplify the definition of decorators (in older versions of Python you need to copy the function attributes `__name__`, `__doc__`, `__module__` and `__dict__` from the original function to the decorated function by hand).

The implementation above works in the sense that the decorator can accept functions with generic signatures; unfortunately this implementation does *not* define a signature-preserving decorator, since in general `memoize25` returns a function with a *different signature* from the original function.

Consider for instance the following case:

```
>>> @memoize25
... def f1(x):
...     time.sleep(1) # simulate some long computation
...     return x
```

Here the original function takes a single argument named `x`, but the decorated function takes any number of arguments and keyword arguments:

```
>>> from inspect import getargspec
>>> print getargspec(f1)
([], 'args', 'kw', None)
```

This means that introspection tools such as pydoc will give wrong informations about the signature of f1. This is pretty bad: pydoc will tell you that the function accepts a generic signature *args, **kw, but when you try to call the function with more than an argument, you will get an error:

```
>>> f1(0, 1)
Traceback (most recent call last):
   ...
TypeError: f1() takes exactly 1 argument (2 given)
```

The solution

The solution is to provide a generic factory of generators, which hides the complexity of making signature-preserving decorators from the application programmer. The decorator function in the decorator module is such a factory:

```
>>> from decorator import decorator
```

decorator takes two arguments, a caller function describing the functionality of the decorator and a function to be decorated; it returns the decorated function. The caller function must have signature (f, *args, **kw) and it must call the original function f with arguments args and kw, implementing the wanted capability, i.e. memoization in this case:

```
def _memoize(func, *args, **kw):
    if kw: # frozenset is used to ensure hashability
        key = args, frozenset(kw.iteritems())
    else:
        key = args
    cache = func.cache # attributed added by memoize
    if key in cache:
        return cache[key]
    else:
        cache[key] = result = func(*args, **kw)
        return result
```

At this point you can define your decorator as follows:

```
def memoize(f):
    f.cache = {}
    return decorator(_memoize, f)
```

The difference with respect to the Python 2.5 approach, which is based on nested functions, is that the decorator module forces you to lift the inner function at the outer level (*flat is better than nested*). Moreover, you are forced to pass explicitly the function you want to decorate to the caller function.

Here is a test of usage:

```
>>> @memoize
... def heavy_computation():
...     time.sleep(2)
...     return "done"

>>> print heavy_computation() # the first time it will take 2 seconds
done

>>> print heavy_computation() # the second time it will be instantaneous
done
```

The signature of `heavy_computation` is the one you would expect:

```
>>> print getargspec(heavy_computation)
([], None, None, None)
```

A `trace` decorator

As an additional example, here is how you can define a trivial `trace` decorator, which prints a message everytime the traced function is called:

```
def _trace(f, *args, **kw):
    print "calling %s with args %s, %s" % (f.__name__, args, kw)
    return f(*args, **kw)

def trace(f):
    return decorator(_trace, f)
```

Here is an example of usage:

```
>>> @trace
... def f1(x):
...     pass
```

It is immediate to verify that `f1` works

```
>>> f1(0)
calling f1 with args (0,), {}
```

and it that it has the correct signature:

```
>>> print getargspec(f1)
(['x'], None, None, None)
```

The same decorator works with functions of any signature:

```
>>> @trace
... def f(x, y=1, z=2, *args, **kw):
...     pass

>>> f(0, 3)
calling f with args (0, 3, 2), {}

>>> print getargspec(f)
(['x', 'y', 'z'], 'args', 'kw', (1, 2))
```

That includes even functions with exotic signatures like the following:

```
>>> @trace
... def exotic_signature((x, y)=(1,2)): return x+y

>>> print getargspec(exotic_signature)
([['x', 'y']], None, None, ((1, 2),))
>>> exotic_signature()
calling exotic_signature with args ((1, 2),), {}
3
```

Notice that the support for exotic signatures has been deprecated in Python 2.6 and removed in Python 3.0.

`decorator` is a decorator

It may be annoying to write a caller function (like the `_trace` function above) and then a trivial wrapper (`def trace(f): return decorator(_trace, f)`) every time. For this reason, the `decorator` module provides an easy shortcut to convert the caller function into a signature-preserving decorator: you can just call `decorator` with a single argument. In our example you can just write `trace = decorator(_trace)`. The `decorator` function can also be used as a signature-changing decorator, just as `classmethod` and `staticmethod`. However, `classmethod` and `staticmethod` return generic objects which are not callable, while `decorator` returns signature-preserving decorators, i.e. functions of a single argument. For instance, you can write directly

```
>>> @decorator
... def trace(f, *args, **kw):
...     print "calling %s with args %s, %s" % (f.func_name, args, kw)
...     return f(*args, **kw)
```

and now `trace` will be a decorator. You can easily check that the signature has changed:

```
>>> print getargspec(trace)
(['f'], None, None, None)
```

Therefore now `trace` can be used as a decorator and the following will work:

```
>>> @trace
... def func(): pass

>>> func()
calling func with args (), {}
```

For the rest of this document, I will discuss examples of useful decorators built on top of `decorator`.

blocking

Sometimes one has to deal with blocking resources, such as `stdin`, and sometimes it is best to have back a "busy" message than to block everything. This behavior can be implemented with a suitable decorator:

```
def blocking(not_avail="Not Available"):
    def _blocking(f, *args, **kw):
        if not hasattr(f, "thread"): # no thread running
            def set_result(): f.result = f(*args, **kw)
            f.thread = threading.Thread(None, set_result)
            f.thread.start()
            return not_avail
        elif f.thread.isAlive():
            return not_avail
        else: # the thread is ended, return the stored result
            del f.thread
            return f.result
    return decorator(_blocking)
```

(notice that without the help of `decorator`, an additional level of nesting would have been needed). This is actually an example of a one-parameter family of decorators.

Functions decorated with `blocking` will return a busy message if the resource is unavailable, and the intended result if the resource is available. For instance:

```
>>> @blocking("Please wait ...")
... def read_data():
...     time.sleep(3) # simulate a blocking resource
...     return "some data"

>>> print read_data() # data is not available yet
Please wait ...

>>> time.sleep(1)
>>> print read_data() # data is not available yet
Please wait ...

>>> time.sleep(1)
>>> print read_data() # data is not available yet
Please wait ...

>>> time.sleep(1.1) # after 3.1 seconds, data is available
>>> print read_data()
some data
```

async

We have just seen an examples of a simple decorator factory, implemented as a function returning a decorator. For more complex situations, it is more convenient to implement decorator factories as classes returning callable objects that can be used as signature-preserving decorators. The suggested pattern to do that is to introduce a helper method `call(self, func, *args, **kw)` and to call it in the `__call__(self, func)` method.

As an example, here I show a decorator which is able to convert a blocking function into an asynchronous function. The function, when called, is executed in a separate thread. Moreover, it is possible to set three callbacks `on_success`, `on_failure` and `on_closing`, to specify how to manage the function call. The implementation is the following:

```
def on_success(result): # default implementation
    "Called on the result of the function"
    return result

def on_failure(exc_info): # default implementation
    "Called if the function fails"
    pass

def on_closing(): # default implementation
    "Called at the end, both in case of success and failure"
    pass
```

```
class Async(object):
    """
    A decorator converting blocking functions into asynchronous
    functions, by using threads or processes. Examples:

    async_with_threads =  Async(threading.Thread)
    async_with_processes =  Async(multiprocessing.Process)
    """

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

    def __call__(self, func, on_success=on_success,
                 on_failure=on_failure, on_closing=on_closing):
        # every decorated function has its own independent thread counter
        func.counter = itertools.count(1)
        func.on_success = on_success
        func.on_failure = on_failure
        func.on_closing = on_closing
        return decorator(self.call, func)

    def call(self, func, *args, **kw):
        def func_wrapper():
            try:
                result = func(*args, **kw)
            except:
                func.on_failure(sys.exc_info())
            else:
                return func.on_success(result)
            finally:
                func.on_closing()
        name = '%s-%s' % (func.__name__, func.counter.next())
        thread = self.threadfactory(None, func_wrapper, name)
        thread.start()
        return thread
```

The decorated function returns the current execution thread, which can be stored and checked later, for instance to verify that the thread `.isAlive()`.

Here is an example of usage. Suppose one wants to write some data to an external resource which can be accessed by a single user at once (for instance a printer). Then the access to the writing function must be locked. Here is a minimalistic example:

```
>>> async = Async(threading.Thread)
```

```
>>> datalist = [] # for simplicity the written data are stored into a list.

>>> @async
... def write(data):
...     # append data to the datalist by locking
...     with threading.Lock():
...         time.sleep(1) # emulate some long running operation
...         datalist.append(data)
...     # other operations not requiring a lock here
```

Each call to `write` will create a new writer thread, but there will be no synchronization problems since `write` is locked.

```
>>> write("data1")
<Thread(write-1, started)>

>>> time.sleep(.1) # wait a bit, so we are sure data2 is written after data1

>>> write("data2")
<Thread(write-2, started)>

>>> time.sleep(2) # wait for the writers to complete

>>> print datalist
['data1', 'data2']
```

The `FunctionMaker` class

You may wonder about how the functionality of the `decorator` module is implemented. The basic building block is a `FunctionMaker` class which is able to generate on the fly functions with a given name and signature from a function template passed as a string. Generally speaking, you should not need to resort to `FunctionMaker` when writing ordinary decorators, but it is handy in some circumstances. We will see an example in two paragraphs, when implementing a custom decorator factory (`decorator_apply`).

Notice that while I do not have plans to change or remove the functionality provided in the `FunctionMaker` class, I do not guarantee that it will stay unchanged forever. For instance, right now I am using the traditional string interpolation syntax for function templates, but Python 2.6 and Python 3.0 provide a newer interpolation syntax and I may use the new syntax in the future. On the other hand, the functionality provided by `decorator` has been there from version 0.1 and it is guaranteed to stay there forever.

`FunctionMaker` takes the name and the signature (as a string) of a function in input, or a whole function. Then, it creates a new function (actually a closure) from a function template (the function template must begin with `def` with no comments before and you cannot use a `lambda`) via its `.make` method: the name and the signature of the resulting function are determinated by the specified name and signature. For instance, here is an example of how to restrict the signature of a function:

```
>>> def f(*args, **kw): # a function with a generic signature
...     print args, kw

>>> fun = FunctionMaker(name="f1", signature="a,b")
>>> f1 = fun.make('''\
... def %(name)s(%(signature)s):
...     f(%(signature)s)''', dict(f=f))
...
>>> f1(1,2)
(1, 2) {}
```

The dictionary passed in this example (`dict(f=f)`) is the execution environment: `FunctionMaker.make` actually returns a closure, and the original function `f` is a variable in the closure environment. `FunctionMaker.make` also accepts keyword arguments and such arguments are attached to the resulting function. This is useful if you want to set some function attributes, for instance the docstring `__doc__`.

For debugging/introspection purposes it may be useful to see the source code of the generated function; to do that, just pass the flag `addsource=True` and a `__source__` attribute will be added to the decorated function:

```
>>> f1 = fun.make('''\
... def %(name)s(%(signature)s):
...     f(%(signature)s)''', dict(f=f), addsource=True)
...
>>> print f1.__source__
def f1(a,b):
    f(a,b)
<BLANKLINE>
```

Getting the source code

Internally `FunctionMaker.make` uses `exec` to generate the decorated function. Therefore `inspect.getsource` will not work for decorated functions. That means that the usual '??' trick in IPython will give you the (right on the spot) message `Dynamically generated function. No source code available`. In the past I have considered this acceptable, since `inspect.getsource` does not really work even with regular decorators. In that case `inspect.getsource` gives you the wrapper source code which is probably not what you want:

```
def identity_dec(func):
    def wrapper(*args, **kw):
        return func(*args, **kw)
    return wrapper
```

```
@identity_dec
def example(): pass

>>> print inspect.getsource(example)
    def wrapper(*args, **kw):
        return func(*args, **kw)
<BLANKLINE>
```

(see bug report 1764286 for an explanation of what is happening). Unfortunately the bug is still there, even in Python 2.6 and 3.0. There is however a workaround. The decorator module adds an attribute `.undecorated` to the decorated function, containing a reference to the original function. The easy way to get the source code is to call `inspect.getsource` on the undecorated function:

```
>>> print inspect.getsource(factorial.undecorated)
@tail_recursive
def factorial(n, acc=1):
    "The good old factorial"
    if n == 0: return acc
    return factorial(n-1, n*acc)
<BLANKLINE>
```

Dealing with third party decorators

Sometimes you find on the net some cool decorator that you would like to include in your code. However, more often than not the cool decorator is not signature-preserving. Therefore you may want an easy way to upgrade third party decorators to signature-preserving decorators without having to rewrite them in terms of `decorator`. You can use a `FunctionMaker` to implement that functionality as follows:

```
def decorator_apply(dec, func):
    "Decorate a function using a signature-non-preserving decorator"
    fun = FunctionMaker(func)
    src = '''def %(name)s(%(signature)s):
    return decorated(%(signature)s)'''
    return fun.make(src, dict(decorated=dec(func)), undecorated=func)
```

`decorator_apply` sets the attribute `.undecorated` of the generated function to the original function, so that you can get the right source code.

Notice that I am not providing this functionality in the `decorator` module directly since I think it is best to rewrite the decorator rather than adding an additional level of indirection. However, practicality beats purity, so you can add `decorator_apply` to your toolbox and use it if you need to.

In order to give an example of usage of `decorator_apply`, I will show a pretty slick decorator that converts a tail-recursive function in an iterative function. I have shamelessly stolen the basic idea from Kay Schluehr's recipe in the Python Cookbook, http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/496691.

```
class TailRecursive(object):
    """
    tail_recursive decorator based on Kay Schluehr's recipe
    http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/496691
    """
    CONTINUE = object() # sentinel

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

    def __call__(self, *args, **kwd):
        try:
            if self.firstcall: # start looping
                self.firstcall = False
                while True:
                    result = self.func(*args, **kwd)
                    if result is self.CONTINUE: # update arguments
                        args, kwd = self.argskwd
                    else: # last call
                        break
            else: # return the arguments of the tail call
                self.argskwd = args, kwd
                return self.CONTINUE
        except: # reset and re-raise
            self.firstcall = True
            raise
        else: # reset and exit
            self.firstcall = True
            return result
```

Here the decorator is implemented as a class returning callable objects.

```
def tail_recursive(func):
    return decorator_apply(TailRecursive, func)
```

Here is how you apply the upgraded decorator to the good old factorial:

```
@tail_recursive
def factorial(n, acc=1):
```

```
    "The good old factorial"
    if n == 0: return acc
    return factorial(n-1, n*acc)

>>> print factorial(4)
24
```

This decorator is pretty impressive, and should give you some food for your mind ;) Notice that there is no recursion limit now, and you can easily compute `factorial(1001)` or larger without filling the stack frame. Notice also that the decorator will not work on functions which are not tail recursive, such as the following

```
def fact(n): # this is not tail-recursive
    if n == 0: return 1
    return n * fact(n-1)
```

(reminder: a function is tail recursive if it either returns a value without making a recursive call, or returns directly the result of a recursive call).

Caveats and limitations

The first thing you should be aware of, it the fact that decorators have a performance penalty. The worse case is shown by the following example:

```
$ cat performance.sh
python -m timeit -s "
from decorator import decorator

@decorator
def do_nothing(func, *args, **kw):
    return func(*args, **kw)

@do_nothing
def f():
    pass
" "f()"

python -m timeit -s "
def f():
    pass
" "f()"
```

On my MacBook, using the `do_nothing` decorator instead of the plain function is more than three times slower:

```
$ bash performance.sh
1000000 loops, best of 3: 0.995 usec per loop
1000000 loops, best of 3: 0.273 usec per loop
```

It should be noted that a real life function would probably do something more useful than `f` here, and therefore in real life the performance penalty could be completely negligible. As always, the only way to know if there is a penalty in your specific use case is to measure it.

You should be aware that decorators will make your tracebacks longer and more difficult to understand. Consider this example:

```
>>> @trace
... def f():
...     1/0
```

Calling `f()` will give you a `ZeroDivisionError`, but since the function is decorated the traceback will be longer:

```
>>> f()
Traceback (most recent call last):
  ...
     File "<string>", line 2, in f
     File "<doctest __main__[18]>", line 4, in trace
       return f(*args, **kw)
     File "<doctest __main__[47]>", line 3, in f
       1/0
ZeroDivisionError: integer division or modulo by zero
```

You see here the inner call to the decorator `trace`, which calls `f(*args, **kw)`, and a reference to `File "<string>", line 2, in f`. This latter reference is due to the fact that internally the decorator module uses `exec` to generate the decorated function. Notice that `exec` is *not* responsibile for the performance penalty, since is the called *only once* at function decoration time, and not every time the decorated function is called.

At present, there is no clean way to avoid `exec`. A clean solution would require to change the CPython implementation of functions and add an hook to make it possible to change their signature directly. That could happen in future versions of Python (see PEP 362) and then the decorator module would become obsolete. However, at present, even in Python 3.0 it is impossible to change the function signature directly, therefore the `decorator` module is still useful. Actually, this is one of the main reasons why I am releasing version 3.0.

In the present implementation, decorators generated by `decorator` can only be used on user-defined Python functions or methods, not on generic callable objects, nor on built-in functions, due to limitations of the `inspect` module in the standard library. Moreover, notice that you can decorate a method, but only before if becomes a bound or unbound method, i.e. inside the class. Here is an example of valid decoration:

```
>>> class C(object):
...     @trace
...     def meth(self):
...         pass
```

Here is an example of invalid decoration, when the decorator in called too late:

```
>>> class C(object):
...     def meth(self):
...         pass
...
>>> trace(C.meth)
Traceback (most recent call last):
  ...
TypeError: You are decorating a non function: <unbound method C.meth>
```

The solution is to extract the inner function from the unbound method:

```
>>> trace(C.meth.im_func)
<function meth at 0x...>
```

There is a restriction on the names of the arguments: for instance, if try to call an argument `_call_` or `_func_` you will get a `NameError`:

```
>>> @trace
... def f(_func_): print f
...
Traceback (most recent call last):
  ...
NameError: _func_ is overridden in
def f(_func_):
    return _call_(_func_, _func_)
```

Finally, the implementation is such that the decorated function contains a *copy* of the original function dictionary (`vars(decorated_f) is not vars(f)`):

```
>>> def f(): pass # the original function
>>> f.attr1 = "something" # setting an attribute
>>> f.attr2 = "something else" # setting another attribute

>>> traced_f = trace(f) # the decorated function

>>> traced_f.attr1
'something'
>>> traced_f.attr2 = "something different" # setting attr
>>> f.attr2 # the original attribute did not change
'something else'
```

Compatibility notes

Version 3.0 is a complete rewrite of the original implementation. It is mostly compatible with the past, a part for a few differences.

First of all, the utilites `get_info` and `new_wrapper`, available in the 2.X versions, have been deprecated and they will be removed in the future. For the moment, using them raises a `DeprecationWarning`. Incidentally, the functionality has been implemented through a decorator which makes a good example for this documentation:

```
@decorator
def deprecated(func, *args, **kw):
    "A decorator for deprecated functions"
    warnings.warn(
        ('Calling the deprecated function %r\n'
         'Downgrade to decorator 2.3 if you want to use this functionality')
        % func.__name__, DeprecationWarning, stacklevel=3)
    return func(*args, **kw)
```

`get_info` has been removed since it was little used and since it had to be changed anyway to work with Python 3.0; `new_wrapper` has been removed since it was useless: its major use case (converting signature changing decorators to signature preserving decorators) has been subsumed by `decorator_apply` and the other use case can be managed with the `FunctionMaker`.

Finally `decorator` cannot be used as a class decorator and the functionality introduced in version 2.3 has been removed. That means that in order to define decorator factories with classes you need to define the `__call__` method explicitly (no magic anymore).

All these changes should not cause any trouble, since they were all rarely used features. Should you have any trouble, you can always downgrade to the 2.3 version.

The examples shown here have been tested with Python 2.5. Python 2.4 is also supported - of course the examples requiring the `with` statement will not work there. Python 2.6 works fine, but if you run the examples here in the interactive interpreter you will notice a couple of minor differences since `getargspec` returns an `ArgSpec` namedtuple instead of a regular tuple, and the string representation of a thread object returns a thread identifier number. That means that running the file `documentation.py` under Python 2.5 will a few errors, but they are not serious. Python 3.0 is kind of supported too. Simply run the script `2to3` on the module `decorator.py` and you will get a version of the code running with Python 3.0 (at least, I did some simple checks and it seemed to work). However there is no support for function annotations yet since it seems premature at this moment (most people are still using Python 2.5).

LICENCE

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

If you use this software and you are happy with it, consider sending me a note, just to gratify my ego. On the other hand, if you use this software and you are unhappy with it, send me a patch!