Advanced methods for creating decorators

Graham Dumpleton

PyCon US - April 2014

Intermission

Rant about the history of this talk and why this topic matters.

Python decorator syntax

```
@function_wrapper
def function():
    pass
```


Even if you have never written your own decorator and have only used them, you would know decorators from the @ symbol used to prefix their usage.

What is happening?

```
def function():
    pass

function = function_wrapper(function)
```


The @ symbol here though is little more than syntactic sugar. One can do the same thing invoking the decorator function explicitly, passing in the function to be decorated and replacing the original with the result. In fact this is what you had to do before the decorator syntax was introduced in Python 2.4.

What is happening?

```
def function():
    pass

function = function_wrapper(function)
```


The decorator syntax is therefore just a short hand way of being able to apply a wrapper around an existing function, or otherwise modify the existing function in place, while the definition of the function is being setup.

Function wrapper

```
class function_wrapper(object):

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

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

@function_wrapper
def function():
    pass
```


The more illustrative way of showing how a wrapper works is to implement it using a class object. The class instance is initialised with and remembers the original function object. When the now wrapped function is called, it is actually the __call__() method of the wrapper object which is invoked. This in turn would then call the original wrapped function.

Doing work in the wrapper

```
class function_wrapper(object):

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

    def __call__(self, *args, **kwargs):
        name = self.wrapped.__name__
        print('enter %s()' % name)
        try:
            return self.wrapped(*args, **kwargs)
        finally:
            print('exit %s()' % name)
```


A pass through wrapper isn't particularly useful, so normally you would actually want to do some work either before or after the wrapped function is called. Or you may want to modify the input arguments or the result as they pass through the wrapper.

Using function closures

```
def function_wrapper(wrapped):

    def _wrapper(*args, **kwargs):
        name = wrapped.__name__
        print('enter %s()' % name)
        try:
            return wrapped(*args, **kwargs)
        finally:
            print('exit %s()' % name)

    return _wrapper

@function_wrapper
def function():
    pass
```


Using a class to implement the wrapper for a decorator isn't actually that popular. Instead a function closure is more often used. In this case a nested function is used as the wrapper and it is that which is returned by the decorator function. When the now wrapped function is called, the nested function is actually being called. This in turn would again then call the original wrapped function.

Using function closures

```
def function_wrapper(wrapped):

    def _wrapper(*args, **kwargs):
        name = wrapped.__name__
        print('enter %s()' % name)
        try:
            return wrapped(*args, **kwargs)
        finally:
            print('exit %s()' % name)

    return _wrapper

@function_wrapper
def function():
    pass
```


In this situation the nested function doesn't actually get passed the original wrapped function explicitly. But it will still have access to it via the arguments given to the outer function call. This does away with the need to create a class to hold what was the wrapped function and thus why it is convenient and generally more popular.

Introspecting functions

```
def function_wrapper(wrapped):
    def _wrapper(*args, **kwargs):
        return wrapped(*args, **kwargs)
    return _wrapper

@function_wrapper
def function():
    pass

>>> print(function.__name__)
_wrapper
```


Now when we talk about functions, we expect them to specify properties which describe them as well as document what they do. These include the __name__ and __doc__ attributes. When we use a wrapper though, this no longer works as we expect as in the case of using a function closure, the details of the nested function are returned.

Class instances do not have names

```
class function_wrapper(object):
    def __init__(self, wrapped):
        self.wrapped = wrapped
    def __call__(self, *args, **kwargs):
        return self.wrapped(*args, **kwargs)

@function_wrapper
def function():
    pass

>>> print(function.__name__)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'function_wrapper' object
    has no attribute '__name__'
```


If we use a class to implement the wrapper, as class instances do not normally have a __name__ attribute, attempting to access the name of the function will actually result in an AttributeError exception.

Copying attributes to the wrapper

```
def function_wrapper(wrapped):
    def _wrapper(*args, **kwargs):
        return wrapped(*args, **kwargs)
    _wrapper.__name__ = wrapped.__name__
    _wrapper.__doc__ = wrapped.__doc__
    return _wrapper

@function_wrapper
def function():
    pass

>>> print(function.__name__)
function
```


The solution here when using a function closure is to copy the attributes of interest from the wrapped function to the nested wrapper function. This will then result in the function name and documentation strings being correct.

Using functools.wraps()

```
import functools

def function_wrapper(wrapped):
    @functools.wraps(wrapped)
    def _wrapper(*args, **kwargs):
        return wrapped(*args, **kwargs)
    return _wrapper

@function_wrapper
def function():
    pass

>>> print(function.__name__)
function
```


Needing to manually copy the attributes is laborious, and would need to be updated if any further special attributes were added which needed to be copied. For example, we should also copy the __module__ attribute, and in Python 3 the __qualname__ and __annotations__ attributes were added. To aid in getting this right, the Python standard library provides the functools.wraps() decorator which does this task for you.

Using functools.update_wrapper()

```
import functools

class function_wrapper(object):

    def __init__(self, wrapped):
        self.wrapped = wrapped
        functools.update_wrapper(self, wrapped)

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


If using a class to implement the wrapper, instead of the functools.wraps() decorator, we would use the functools.update_wrapper() function.

Function argument specifications

```
import inspect

def function_wrapper(wrapped): ...

@function_wrapper
def function(arg1, arg2): pass

>>> print(inspect.getargspec(function))
ArgSpec(args=[], varargs='args',
    keywords='kwargs', defaults=None)
```


So we have managed to fix things up so the function name and any documentation string is correct, but what if we want to query the argument specification. This also fails and instead of returning the argument specification for the wrapped function, it returns that of the wrapper. In the case of using a function closure, this is the nested function. The decorator is therefore not signature preserving.

Class instances are not wrappers.

```
class function_wrapper(object): ...

@function_wrapper
def function(arg1, arg2): pass

>>> print(inspect.getargspec(function))
Traceback (most recent call last):
  File "...", line XXX, in <module>
    print(inspect.getargspec(function))
  File ".../inspect.py", line 813, in getargspec
    raise TypeError('{!r} is not a Python
        function'.format(func))
TypeError: <__main__.function_wrapper object at
        0x107e0ac90> is not a Python function
```


A worse situation again occurs with the class wrapper. This time we get an exception complaining that the wrapped function isn't actually a function. As a result it isn't possible to derive an argument specification, even though the wrapped function is actually still callable.

Functions vs Methods

```
class Class(object):

    @function_wrapper
    def method(self):
        pass

    @classmethod
    def cmethod(cls):
        pass

    @staticmethod
    def smethod():
        pass
```


Now, as well as normal functions, decorators can also be applied to methods of classes. Python even includes a couple of special decorators called @classmethod and @staticmethod for converting normal instance methods into these special method types. Methods of classes do provide a number of potential problems though.

Wrapping class and static methods

```
class Class(object):

    @function_wrapper
    @classmethod
    def cmethod(cls):
        pass
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in Class
  File "<stdin>", line 2, in wrapper
  File ".../functools.py", line 33, in update_wrapper
    setattr(wrapper, attr, getattr(wrapped, attr))
AttributeError: 'classmethod' object has no
        attribute '__module__'
```


The first is that even if using functools.wraps() or functools.update_wrapper() in your decorator, when the decorator is applied around @classmethod or @staticmethod, it will fail with an exception. This is because the wrappers created by these, do not have some of the attributes being copied. As it happens, this is a Python 2 bug and it is fixed in Python 3 by ignoring missing attributes.

Wrappers aren't always callable

```
class Class(object):
    @function_wrapper
    @classmethod
    def cmethod(cls):
        pass
```

```
Class.cmethod()

Traceback (most recent call last):
  File "classmethod.py", line 15, in <module>
    Class.cmethod()
  File "classmethod.py", line 6, in _wrapper
    return wrapped(*args, **kwargs)
TypeError: 'classmethod' object is not callable
```


Even when we run it under Python 3, we still hit trouble though. This is because both wrapper types assume that the wrapped function is directly callable. This need not actually be the case. A wrapped function can actually be what is called a descriptor, meaning that in order to get back a callable, the descriptor has to be correctly bound to the instance first.

Issues encountered so far

- Preservation of function __name__ and __doc__.
- Preservation of function argument specification.
- Ability to apply decorators on top of other decorators that are implemented as descriptors.

So although decorators using function closures or class wrappers may appear to solve the task at hand, they fail in various corner cases and also don't do a very good job at preserving the ability to do introspection. The latter is a problem for documentation tools, IDEs and also some performance monitoring or profiling tools.

What are descriptors?

. obj.attribute
    --> attribute.__get__(obj, type(obj))

. obj.attribute = value
    --> attribute.__set__(obj, value)

. del obj.attribute
    --> attribute.__delete__(obj)

Lets go back now and look at these descriptors, as they turn out to be a key mechanism in all this. A descriptor is an object attribute with “binding behaviour”, one whose attribute access has been overridden by methods in the descriptor protocol. Those methods are __get__(), __set__(), and __delete__(). If any of those methods are defined for an object, it is said to be a descriptor.

What are descriptors?

- obj.attribute
  --> attribute.__get__(obj, type(obj))

- obj.attribute = value
  --> attribute.__set__(obj, value)

- del obj.attribute
  --> attribute.__delete__(obj)

What this means is that if an attribute of a class has any of these special methods defined, when the corresponding operation is performed on that attribute of a class, then those methods will be called instead of the default action. This allows an attribute to override how those operations are going to work.

Functions are descriptors

```
def f(obj): pass

>>> hasattr(f, '__get__')
True

>>> f
<function f at 0x10e963cf8>

>>> obj = object()

>>> f.__get__(obj, type(obj))
<bound method object.f of <object object
          at 0x10e8ac0b0>>
```


You may well be thinking that you have never made use of descriptors, but fact is that function objects are actually descriptors. When a function is originally added to a class definition it is as a normal function. When you access that function using a dotted attribute path, you are invoking the __get__() method to bind the function to the class instance, turning it into a bound method of that object.

Functions are descriptors

```
def f(obj): pass

>>> hasattr(f, '__get__')
True

>>> f
<function f at 0x10e963cf8>

>>> obj = object()

>>> f.__get__(obj, type(obj))
<bound method object.f of <object object
          at 0x10e8ac0b0>>
```


So when calling a method of a class, it is not the __call__() method of the original function object that is called, but the __call__() method of the temporary bound object that is created as a result of accessing the function. The problem with classmethod was that it is dependent on the descriptor protocol being applied as the __call__() method only exists on the result returned by __get__() when it is called.

Wrappers as descriptors

```
class bound_function_wrapper(object):
    def __init__(self, wrapped):
        self.wrapped = wrapped
    def __call__(self, *args, **kwargs):
        return self.wrapped(*args, **kwargs)
```

```
class function_wrapper(object):
    def __init__(self, wrapped):
        self.wrapped = wrapped
    def __get__(self, instance, owner):
        wrapped = self.wrapped.__get__(
                instance, owner)
        return bound_function_wrapper(wrapped)
    def __call__(self, *args, **kwargs):
        return self.wrapped(*args, **kwargs)
```


The way to solve this is for wrappers to also be descriptors. If the wrapper is applied to a normal function, the __call__() method of the wrapper is used. If the wrapper is applied to a method of a class, the __get__() method is called, which returns a new bound wrapper and the __call__() method of that is invoked instead. This allows our wrapper to be used around descriptors as it propagates the descriptor protocol.

Wrappers as descriptors

```
class bound_function_wrapper(object):
    def __init__(self, wrapped):
        self.wrapped = wrapped
    def __call__(self, *args, **kwargs):
        return self.wrapped(*args, **kwargs)
```

```
class function_wrapper(object):
    def __init__(self, wrapped):
        self.wrapped = wrapped
    def __get__(self, instance, owner):
        wrapped = self.wrapped.__get__(
                  instance, owner)
        return bound_function_wrapper(wrapped)
    def __call__(self, *args, **kwargs):
        return self.wrapped(*args, **kwargs)
```


So since using a function closure will ultimately fail if used around a decorator which is implemented as a descriptor, the situation we therefore have is that if we want everything to work, then decorators should always use this pattern. The question now is how do we address the other issues we had.

What does functools.wraps() do?

```
WRAPPER_ASSIGNMENTS = ('__module__',
        '__name__', '__qualname__', '__doc__',
        '__annotations__')

WRAPPER_UPDATES = ('__dict__',)

def update_wrapper(wrapper,wrapped,
        assigned = WRAPPER_ASSIGNMENTS,
        updated = WRAPPER_UPDATES):

    ...
```


We solved naming using functools.wrap()/ functools.update_wrapper() before, but what do they do. Well wraps() just uses update_wrapper(), so we just need to look at it. I'll show what is in Python 3.3, although that actually has a bug in it, which is fixed in Python 3.4. :-)

What does functools.wraps() do?

```
WRAPPER_ASSIGNMENTS = ('__module__',
        '__name__', '__qualname__', '__doc__',
        '__annotations__')

WRAPPER_UPDATES = ('__dict__',)

def update_wrapper(wrapper,wrapped,
        assigned = WRAPPER_ASSIGNMENTS,
        updated = WRAPPER_UPDATES):

    ...
```


Key thing to try and remember as we try and look at the body of update_wrapper() is what is in these default variables that get passed as 'assigned' and 'updated'. Those in 'assigned' are what we were originally manually assigning, plus some extras. The '__dict__' in 'updates' is something new though so we need to see what is happening with it.

It potentially does lots of copying

```
    wrapper.__wrapped__ = wrapped

    for attr in assigned:
        try:
            value = getattr(wrapped, attr)
        except AttributeError:
            pass
        else:
            setattr(wrapper, attr, value)

    for attr in updated:
        getattr(wrapper, attr).update(
                getattr(wrapped, attr, {}))
```


Looking at the body of the function, three things are being done. First off a reference to the wrapped function is saved as __wrapped__. This is the bug, as it should be done last. The second is to copy those attributes such as __name__ and __doc__. Finally the third thing is to copy the contents of __dict__ from the wrapped function into the wrapper, which could actually result in quite a lot of objects needing to be copied.

Using functools.wraps() is too slow

```
class bound_function_wrapper(object):

    def __init__(self, wrapped):
        self.wrapped = wrapped
        functools.update_wrapper(self, wrapped)
```

```
class function_wrapper(object):

    def __init__(self, wrapped):
        self.wrapped = wrapped
        functools.update_wrapper(self, wrapped)
```


If we are using a function closure or straight class wrapper this copying is able to be done at the point that the decorator is applied. With the wrapper being a descriptor though, it technically now also needs to be done in the bound wrapper. As the bound wrapper is created every time the wrapper is called for a function bound to a class, this is going to be too slow. We need a more performant way of handling this.

Transparent object proxy

```
class object_proxy(object):

    def __init__(self, wrapped):
        self.wrapped = wrapped
        try:
            self.__name__ = wrapped.__name__
        except AttributeError:
            pass

    @property
    def __class__(self):
        return self.wrapped.__class__

    def __getattr__(self, name):
        return getattr(self.wrapped, name)
```


The solution is what is called an object proxy. This is a special wrapper class which looks and behaves like what it wraps. It is a complicated beast in its own right, so I am going to gloss over the details. In short though, it copies limited attributes from the wrapped object to itself, and otherwise uses special methods, properties and __getattr__() to fetch attributes from the wrapped object only when required.

Use object proxy as base class

```
class bound_function_wrapper(object_proxy):

    def __init__(self, wrapped):
        super(bound_function_wrapper,
                self).__init__(wrapped)

class function_wrapper(object_proxy):

    def __init__(self, wrapped):
        super(function_wrapper,
                self).__init__(wrapped)
```


What we now do is derive our wrapper class from the object proxy. Doing so, attributes like __name__ and __doc__, when queried from the wrapper, return the values from the wrapped function instead. Calls like inspect.getargspec() and inspect.getsource() will also work and return what we expect.

A decorator for creating decorators

```
@decorator
def my_function_wrapper(wrapped, args, kwargs):
    return wrapped(*args, **kwargs)

@my_function_wrapper
def function():
    pass
```


So we have a pattern now for implementing a decorator that appears to work correctly, but needing to do all that each time is more work than we really want. What we can do therefore is create a decorator to help us create decorators. This would reduce the code we need to write for each decorator to a single function as shown. What would this decorator factory need to look like?

The decorator factory

```
def decorator(wrapper):
    @functools.wraps(wrapper)
    def _decorator(wrapped):
        return function_wrapper(wrapped, wrapper)
    return _decorator
```


As it turns out, our decorator factory is quite simple and isn't really much different to using a partial(), combining our new wrapper function argument from when the decorator is defined, with the wrapped function when the decorator is used and passing them into our function wrapper object.

Delegating function wrapper

```
class function_wrapper(object_proxy):

    def __init__(self, wrapped, wrapper):
        super(function_wrapper,
                self).__init__(wrapped)
        self.wrapper = wrapper

    def __get__(self, instance, owner):
        wrapped = self.wrapped.__get__(
            instance, owner)
        return bound_function_wrapper(wrapped,
                self.wrapper)

    def __call__(self, *args, **kwargs):
        return self.wrapper(self.wrapped, args,
                kwargs)
```


The __call__() method of our function wrapper, for when the wrapper is used around a normal function, now just calls the decorator wrapper function with the wrapped function and arguments, leaving the calling of the wrapped function up to the decorator wrapper function. In the case where binding a function, the wrapper is also passed to the bound wrapper.

Delegating bound wrapper

```
class bound_function_wrapper(object_proxy):

    def __init__(self, wrapped, wrapper):
        super(bound_function_wrapper,
                self).__init__(wrapped)
        self.wrapper = wrapper

    def __call__(self, *args, **kwargs):
        return self.wrapper(self.wrapped, args,
                kwargs)
```


The bound wrapper is more or less the same, with the __call__() method delegating to the decorator wrapper function. So we can make creating decorators easier using a factory, lets see now what other problems we can find.

Functions vs Methods

```
@decorator
def my_function_wrapper(wrapped, args, kwargs):
    print('ARGS', args)
    return wrapped(*args, **kwargs)
```

```
@my_function_wrapper
def function(a, b):
    pass

>>> function(1, 2)
ARGS (1, 2)
```


The first such issue is creating decorators that can work on both normal functions and instance methods of classes. Changing our decorator to print out the args passed for a normal function we obviously just get a tuple of the two arguments.

Instance methods

```
class Class(object):

    @my_function_wrapper
    def function_im(self, a, b):
        pass

c = Class()

>>> c.function_im()
ARGS (1, 2)
```


Do the same for an instance method and the result is the same. The problem here is what if the decorator wanted to know what the actual instance of the class was? We have lost that information when the function was bound to the class as it is now associated with the wrapped function passed in, rather than the argument list.

Remembering the instance

```
class function_wrapper(object_proxy):

    def __init__(self, wrapped, wrapper):
        super(function_wrapper,
                self).__init__(wrapped)
        self.wrapper = wrapper

    def __get__(self, instance, owner):
        wrapped = self.wrapped.__get__(
            instance, owner)
        return bound_function_wrapper(wrapped,
                instance, self.wrapper)

    def __call__(self, *args, **kwargs):
        return self.wrapper(self.wrapped, None,
                args, kwargs)
```


To solve this problem we can remember what the instance was that was passed to the __get__() method when it was called to bind the function. This can then be passed through to the bound wrapper when it is created.

Passing the instance to the wrapper

```
class bound_function_wrapper(object_proxy):

    def __init__(self, wrapped, instance,
                 wrapper):
        super(bound_function_wrapper,
                self).__init__(wrapped)
        self.instance = instance
        self.wrapper = wrapper

    def __call__(self, *args, **kwargs):
        return self.wrapper(self.wrapped,
                self.instance, args, kwargs)
```


In the bound wrapper, the instance pointer can then be passed through to the decorator wrapper function as an extra argument. You may have missed it, but to be uniform for the case of a normal function, in the top level wrapper we passed None for this new instance argument.

Distinguishing instance method

```
@decorator
def my_function_wrapper(wrapped, instance,
        args, kwargs):
    print('INSTANCE', instance)
    print('ARGS', args)
    return wrapped(*args, **kwargs)
```

```
>>> function(1, 2)
INSTANCE None
ARGS (1, 2)
```

```
>>> c.function_im(1, 2)
INSTANCE <__main__.Class object at 0x1085ca9d0>
ARGS (1, 2)
```


This then allows us to be able to distinguish between a normal function call and an instance method call within the one decorator wrapper function. The reference to the instance is even passed separately so we don't have to juggle with the arguments to move it out of the way for an instance method.

Calling instance method via class

```
>>> Class.function_im(c, 1, 2)
INSTANCE None
ARGS (<__main__.Class object at 0x1085ca9d0>,
        1, 2)
```


Unfortunately we aren't done though, as when calling an instance method via the class, passing in the instance as an argument, the instance passed to the decorator wrapper function is None. Instead the reference to the instance gets passed through as the first argument.

Special case calling via class

```
class bound_function_wrapper(object_proxy):

    def __call__(self, *args, **kwargs):

        if self.instance is None:
            instance, args = args[0], args[1:]
            wrapped = functools.partial(
                    self.wrapped, instance)
            return self.wrapper(wrapped,
                    instance, args, kwargs)

        return self.wrapper(self.wrapped,
                self.instance, args, kwargs)
```


To deal with this variation, we can check for instance being None before calling the decorator wrapper function and pop the instance off the start of the argument list. We then use a partial to bind the instance to the wrapped function ourselves and call the decorator wrapper function. We then get the same result no matter whether the instance method is called via the class or not.

But we broke class methods

```
class Class(object):

    @my_function_wrapper
    @classmethod
    def function_cm(cls, a, b):
        pass
```

```
>>> Class.function_cm(1, 2)
INSTANCE 1
ARGS (2,)
```


This fiddle does though upset things for when we have a class method, also causing the same issue for a static method. In both those cases the instance is also passed as None. The result is that the real first argument ends up as the instance, which is obviously going to be quite wrong.

Split out class and static method

```
class function_wrapper(object_proxy):

    def __get__(self, instance, owner):
        wrapped = self.wrapped.__get__(
                instance, owner)

        if isinstance(self.wrapped,
                (classmethod, staticmethod)):
            bound_type = bound_function_wrapper
        else:
            bound_type = bound_method_wrapper

        return bound_type(wrapped, instance,
                self.wrapper)
```


We can handle that in the top level wrapper by looking at the type of the wrapped function prior to doing binding. If it is a class method or static method, then we know anything else is likely to be an instance method. For a class or static method we use the original bound function wrapper before the fiddle was added and move the fiddle into a version of the wrapper specifically for instance methods.

But the instance is still None

```
class Class(object):

    @my_function_wrapper
    @classmethod
    def function_cm(cls, a, b):
        pass

>>> Class.function_cm(1, 2)
INSTANCE None
ARGS (1, 2)
```


We are still not quite there though. The argument list is right again, but the instance is still None. For a static method this is probably quite reasonable since it isn't really much different to a normal function. For a class method, it would be nice for the instance to actually be the class type corresponding to the initial 'cls' argument for the class method. The big question is whether there is another way of getting this.

The bound class method knows all

```
class bound_function_wrapper(object_proxy):

    def __call__(self, *args, **kwargs):

        instance = getattr(self.wrapped,
                '__self__', None)

        return self.wrapper(self.wrapped,
                instance, args, kwargs)
```


Turns out there is a way of still getting the class the class method is bound to. This is by accessing the __self__ attribute of the bound function. We therefore ignore the instance the __get__() method was passed and use the __self__ attribute instead.

Class and static seen as different

```
>>> c.function_im(1, 2)
INSTANCE <__main__.Class object at 0x1085ca9d0>
ARGS (1, 2)

>>> Class.function_cm(1, 2)
INSTANCE <class '__main__.Class'>
ARGS (1, 2)

>>> Class.function_sm(1, 2)
INSTANCE None
ARGS (1, 2)
```


Success, finally. We now have the instance argument for an instance method being the instance of the class. For a class method it is the class itself, and for a normal function, the instance is None.

Decorating a class

```
@my_function_wrapper
class Class(object): ...

>>> c = Class()
WRAPPED <class '__main__.Class'>
INSTANCE None
ARGS ()
```


We have one more situation to consider though. That is where we want to decorate a class. What happens then? In this case the instance is still None, so from that we cannot distinguish it from a normal function. If we also look at the wrapped function though, we will see that it is a class type, where as it would be a function in the case of a normal function being called.

Universal decorator

```
@decorator
def universal(wrapped, instance, args, kwargs):
    if instance is None:
        if inspect.isclass(wrapped):
            # class.
        else:
            # function or staticmethod.
    else:
        if inspect.isclass(instance):
            # classmethod.
        else:
            # instancemethod.
```


This works out okay though, because we can look at the type of what is being wrapped in that case. This means we now have the ability to create a universal decorator. That is, a decorator that can determine what it is wrapping. This does away with the need to create separate decorators for functions and instance methods which would otherwise do the same thing.

Required decorator arguments

```
def required_arguments(arg):
    @decorator
    def _wrapper(wrapped, instance, args, kwargs):
        return wrapped(*args, **kwargs)
    return _wrapper

@required_arguments(arg=1)
def function():
    pass
```


Now, the decorators so far did not allow arguments to be supplied when being applied to a function. If arguments to the decorator are required, the decorator definition can be nested within a function to create a function closure. When the outer decorator factory function is used, it returns the inner decorator function. Positional or keyword arguments can be used, but keyword arguments are possibly a better convention.

Optional decorator arguments

```
def optional_arguments(wrapped=None, arg=1):
    if wrapped is None:
        return functools.partial(
            optional_arguments, arg=arg)

    @decorator
    def _wrapper(wrapped, instance, args, kwargs):
        return wrapped(*args, **kwargs)
    return _wrapper(wrapped)

@optional_arguments(arg=2)
def function1():
    pass
```


If arguments have default values, the outer decorator factory would take the wrapped function as first argument with None as a default. The decorator arguments follow. Decorator arguments would now be passed as keyword arguments. On the first call, wrapped will be None, and a partial is used to return the decorator factory again. On the second call, wrapped is passed and this time it is wrapped with the decorator.

No need to supply arguments

```
def optional_arguments(wrapped=None, arg=1):
    if wrapped is None:
        return functools.partial(
            optional_arguments, arg=arg)

    @decorator
    def _wrapper(wrapped, instance, args, kwargs):
        return wrapped(*args, **kwargs)
    return _wrapper(wrapped)

@optional_arguments
def function2():
    pass
```


Because we have default arguments though, we don't actually need to pass the arguments, in which case the decorator factory is applied direct to the function being decorated. Because wrapped is not None when passed in, the decorator is wrapped around the function immediately, skipping the return of the factory a second time.

Forcing keyword arguments

```
def required_arguments(*, arg):
    @decorator
    def _wrapper(wrapped, instance, args, kwargs):
        return wrapped(*args, **kwargs)
    return _wrapper

def optional_arguments(wrapped=None, *, arg=1):
    if wrapped is None:
        return functools.partial(
            optional_arguments, arg=arg)

    @decorator
    def _wrapper(wrapped, instance, args, kwargs):
        return wrapped(*args, **kwargs)
    return _wrapper(wrapped)
```


Now why I said a convention of having keyword arguments is preferable, is that Python 3 allows you to enforce it using the new keyword only argument syntax. This way you avoid the problem of someone passing in a decorator argument as the positional argument for wrapped. For consistency, keyword only arguments can also be enforced for required arguments even though it isn't strictly necessary.

It can do everything

. Preserves __name__.
. Preserves __doc__.
. Signature from inspect.getargspec() is correct.
. Source code from inspect.getsource() is correct.
. Decorators can be aware of the context they are used in.
. Decorator arguments are easily supported.
. Optional decorator arguments are also possible.

The final result is that we now have a means of creating decorators that preserves the function name, documentation strings, argument specification and even retrieving of source code. One decorator can be used on classes, functions, instance methods and class methods. It is also easy to support decorator arguments, even allowing them to be optional if desired.

Want more detail?

- Decorator blog post series.
  - blog: http://blog.dscpl.com.au/search/label/decorators
  - index: https://github.com/GrahamDumpleton/wrapt/tree/master/blog

This has been a whirlwind tour of this topic. As much as I covered it still doesn't actually cover everything I could. As I wasn't expecting to be doing this talk here at PyCon, in January this year I started a more in depth series of blog posts on this topic. You can find these on my blog site, or via the more easily parsed index I have created on github. Although I have paused for now, expect more posts in this series.

Use the 'wrapt' package

- wrapt - A Python module for decorators, wrappers and monkey patching.
  - docs: http://wrapt.readthedocs.org/
  - github: https://github.com/GrahamDumpleton/wrapt
  - pypi: https://pypi.python.org/pypi/wrapt

Graham.Dumpleton@gmail.com
@GrahamDumpleton

So as you probably already knew, decorators are a quite simple concept. Or at least should I say that they can be made to be simple to use. The actual work involved in getting them to work properly is a lot more. Rather than replicate all of what I discussed, I have created a package that bundles up all this magic. This package is called 'wrapt' and you can find source code on github and install it from PyPi.