# The `decorator` module


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



| | |
|---|---|
| **Version:** | 4.1.0 (2017-07-15) |
| **Supports:** | Python 2.6, 2.7, 3.0, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6 |
| **Download page:** | http://pypi.python.org/pypi/decorator/4.1.0 |
| **Installation:** | `pip install decorator` |
| **License:** | BSD license |


## Contents

# Introduction

The `decorator` module is over ten years old, but still alive and kicking. It is used by several frameworks (IPython, scipy, authkit, pylons, pycuda, sugar, ...) and has been stable for a *long* time. It is your best option if you want to preserve the signature of decorated functions in a consistent way across Python releases. Version 4.0 is fully compatible with the past, except for one thing: support for Python 2.4 and 2.5 has been dropped. That decision made it possible to use a single code base both for Python 2.X and Python 3.X. This is a *huge* bonus, since I could remove over 2,000 lines of duplicated documentation/doctests. Having to maintain separate docs for Python 2 and Python 3 effectively stopped any development on the module for several years. Moreover, it is now trivial to distribute the module as an universal wheel since 2to3 is no more required. Since Python 2.5 has been released 9 years ago, I felt that it was reasonable to drop the support for it. If you need to support ancient versions of Python, stick with the decorator module version 3.4.2. The current version supports all Python releases from 2.6 up to 3.6.

# What's New in version 4

- **New documentation** There is now a single manual for all Python versions, so I took the opportunity to overhaul the documentation. So, even if you are a long-time user, you may want to revisit the docs, since several examples have been improved.
- **Packaging improvements** The code is now also available in wheel format. Integration with setuptools has improved and you can run the tests with the command `python setup.py test` too.
- **Code changes** A new utility function `decorate(func, caller)` has been added. It does the same job that was performed by the older `decorator(caller, func)`. The old functionality is now deprecated and no longer documented, but still available for now.
- **Multiple dispatch** The decorator module now includes an implementation of generic functions (sometimes called "multiple dispatch functions"). The API is designed to mimic `functools.singledispatch` (added in Python 3.4), but the implementation is much simpler. Moreover, all decorators involved preserve the signature of the decorated functions. For now, this exists mostly to demonstrate the power of the module. In the future it could be enhanced/optimized; however, its API could change. (Such is the fate of experimental features!) In any case, it is very short and compact (less then 100 lines), so you can extract it for your own use. Take it as food for thought.
- **Python 3.5 coroutines** From version 4.1 it is possible to decorate coroutines, i.e. functions defined with the *async def* syntax, and to mantain the *inspect.iscoroutinefunction* check working for the decorated function.

# Usefulness of decorators

Python decorators are an interesting example of why syntactic sugar matters. In principle, their introduction in Python 2.4 changed nothing, since they did 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 the documentation you are reading 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:**

> Callable objects which accept a function as input and return a function as output, *with the same signature*.

***signature-changing* decorators:**

> Decorators which change the signature of their input function, or decorators that return non-callable objects.

**Signature-changing** decorators have their use: for instance, the builtin classes `staticmethod` and `classmethod` are in this group. They take functions and return descriptor objects which are neither functions, nor callables.

Still, **signature-preserving** decorators are more common, and easier to reason about. In particular, they can be composed together, whereas other decorators generally 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. In recent versions of Python you can find a sophisticated `lru_cache` decorator in the standard library's `functools`. Here I am just interested in giving an example.

Consider the following simple implementation (note that it is generally impossible to *correctly* memoize something that depends on non-hashable arguments):

```
def memoize_uw(func):
    func.cache = {}

    def memoize(*args, **kw):
        if kw:  # frozenset is used to ensure hashability
            key = args, frozenset(kw.items())
        else:
            key = args
        if key not in func.cache:
            func.cache[key] = func(*args, **kw)
```

```
        return func.cache[key]
    return functools.update_wrapper(memoize, func)
```

Here I used the functools.update_wrapper utility, which was added in Python 2.5 to simplify the writing of decorators. (Previously, you needed to manually copy the function attributes `__name__`, `__doc__`, `__module__`, and `__dict__` to the decorated function by hand.)

Here is an example of usage:

```
@memoize_uw
def f1(x):
    "Simulate some long computation"
    time.sleep(1)
    return x
```

This works insofar as the decorator accepts functions with generic signatures. Unfortunately, it is *not* a signature-preserving decorator, since `memoize_uw` generally returns a function with a *different signature* from the original.

Consider for instance the following case:

```
@memoize_uw
def f1(x):
    "Simulate some long computation"
    time.sleep(1)
    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 decorator import getargspec  # akin to inspect.getargspec
>>> print(getargspec(f1))
ArgSpec(args=[], varargs='args', varkw='kw', defaults=None)
```

This means that introspection tools (like `pydoc`) will give false information about the signature of `f1` -- unless you are using Python 3.5. This is pretty bad: `pydoc` will tell you that the function accepts the generic signature `*args, **kw`, but calling the function with more than one argument raises an error:

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

Notice that `inspect.getargspec` and `inspect.getfullargspec` will give the wrong signature. This even occurs in Python 3.5, although both functions were deprecated in that release.

# 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 `decorate` function in the `decorator` module is such a factory:

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

`decorate` takes two arguments:

1. a caller function describing the functionality of the decorator, and
2. a function to be decorated.

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 (in this case, memoization):

```
def _memoize(func, *args, **kw):
    if kw:  # frozenset is used to ensure hashability
        key = args, frozenset(kw.items())
    else:
        key = args
    cache = func.cache  # attribute added by memoize
    if key not in cache:
        cache[key] = func(*args, **kw)
    return cache[key]
```

Now, you can define your decorator as follows:

```
def memoize(f):
    """
    A simple memoize implementation. It works by adding a .cache dictionary
    to the decorated function. The cache will grow indefinitely, so it is
    your responsibility to clear it, if needed.
    """
    f.cache = {}
    return decorate(f, _memoize)
```

The difference from the nested function approach of `memoize_uw` is that the decorator module forces you to lift the inner function to the outer level. Moreover, you are forced to explicitly pass the function you want to decorate; there are no closures.

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))
ArgSpec(args=[], varargs=None, varkw=None, defaults=None)
```

# A `trace` decorator

Here is an example of how to define a simple `trace` decorator, which prints a message whenever the traced function is called:

```
def _trace(f, *args, **kw):
    kwstr = ', '.join('%r: %r' % (k, kw[k]) for k in sorted(kw))
    print("calling %s with args %s, {%s}" % (f.__name__, args, kwstr))
    return f(*args, **kw)
```

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

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))
ArgSpec(args=['x'], varargs=None, varkw=None, defaults=None)
```

The 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))
ArgSpec(args=['x', 'y', 'z'], varargs='args', varkw='kw', defaults=(1, 2))
```

# Function annotations

Python 3 introduced the concept of function annotations: the ability to annotate the signature of a function with additional information, stored in a dictionary named `__annotations__`. The `decorator` module (starting from release 3.3) will understand and preserve these annotations.

Here is an example:

```
>>> @trace
... def f(x: 'the first argument', y: 'default argument'=1, z=2,
...       *args: 'varargs', **kw: 'kwargs'):
...     pass
```

In order to introspect functions with annotations, one needs the utility `inspect.getfullargspec` (introduced in Python 3, then deprecated in Python 3.5, in favor of `inspect.signature`):

```
>>> from inspect import getfullargspec
>>> argspec = getfullargspec(f)
>>> argspec.args
['x', 'y', 'z']
>>> argspec.varargs
'args'
>>> argspec.varkw
'kw'
>>> argspec.defaults
(1, 2)
>>> argspec.kwonlyargs
[]
>>> argspec.kwonlydefaults
```

You can check that the `__annotations__` dictionary is preserved:

```
>>> f.__annotations__ is f.__wrapped__.__annotations__
True
```

Here `f.__wrapped__` is the original undecorated function. This attribute exists for consistency with the behavior of `functools.update_wrapper`.

Another attribute copied from the original function is `__qualname__`, the qualified name. This attribute was introduced in Python 3.3.

# `decorator.decorator`

It can become tedious to write a caller function (like the above `_trace` example) and then a trivial wrapper (`def trace(f): return decorate(f, _trace)`) every time. Not to worry! The `decorator` module provides an easy shortcut to convert the caller function into a signature-preserving decorator.

It is the `decorator` function:

```
>>> from decorator import decorator
>>> print(decorator.__doc__)
decorator(caller) converts a caller function into a decorator
```

The `decorator` function can be used as a signature-changing decorator, just like `classmethod` and `staticmethod`. But `classmethod` and `staticmethod` return generic objects which are not callable. Instead, `decorator` returns signature-preserving decorators (i.e. functions with a single argument).

For instance, you can write:

```
>>> @decorator
... def trace(f, *args, **kw):
...     kwstr = ', '.join('%r: %r' % (k, kw[k]) for k in sorted(kw))
...     print("calling %s with args %s, {%s}" % (f.__name__, args, kwstr))
...     return f(*args, **kw)
```

And `trace` is now a decorator!

```
>>> trace
<function trace at 0x...>
```

Here is an example of usage:

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

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

# blocking

Sometimes one has to deal with blocking resources, such as `stdin`. Sometimes it is better to receive a "busy" message than just blocking everything. This can be accomplished with a suitable family of decorators, where the parameter is the busy message:

```
def blocking(not_avail):
    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)
```

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
```

## `decorator(cls)`

The `decorator` facility can also produce a decorator starting from a class with the signature of a caller. In such a case the produced generator is able to convert functions into factories to create instances of that class.

As an example, here is a decorator which can convert a blocking function into an asynchronous function. When the function is called, it is executed in a separate thread.

(This is similar to the approach used in the `concurrent.futures` package. But I don't recommend that you implement futures this way; this is just an example.)

```
class Future(threading.Thread):
    """
    A class converting blocking functions into asynchronous
    functions by using threads.
    """
    def __init__(self, func, *args, **kw):
        try:
            counter = func.counter
        except AttributeError:  # instantiate the counter at the first call
            counter = func.counter = itertools.count(1)
        name = '%s-%s' % (func.__name__, next(counter))

        def func_wrapper():
            self._result = func(*args, **kw)
        super(Future, self).__init__(target=func_wrapper, name=name)
        self.start()

    def result(self):
        self.join()
        return self._result
```

The decorated function returns a `Future` object. It has a `.result()` method which blocks until the underlying thread finishes and returns the final result.

Here is the minimalistic usage:

```
>>> @decorator(Future)
... def long_running(x):
...     time.sleep(.5)
...     return x

>>> fut1 = long_running(1)
>>> fut2 = long_running(2)
>>> fut1.result() + fut2.result()
3
```

# contextmanager

Python's standard library has the `contextmanager` decorator, which converts a generator function into a `GeneratorContextManager` factory. For instance, if you write this...

```
>>> from contextlib import contextmanager
>>> @contextmanager
... def before_after(before, after):
```

```
...     print(before)
...     yield
...     print(after)
```

...then `before_after` is a factory function that returns `GeneratorContextManager` objects, which provide the use of the `with` statement:

```
>>> with before_after('BEFORE', 'AFTER'):
...     print('hello')
BEFORE
hello
AFTER
```

Basically, it is as if the content of the `with` block was executed in the place of the `yield` expression in the generator function.

In Python 3.2, `GeneratorContextManager` objects were enhanced with a `__call__` method, so that they can be used as decorators, like so:

```
>>> @ba
... def hello():
...     print('hello')
...
>>> hello()
BEFORE
hello
AFTER
```

The `ba` decorator basically inserts a `with ba:` block inside the function.

However, there are two issues:

1. `GeneratorContextManager` objects are only callable in Python 3.2, so the previous example breaks in older versions of Python. (You can solve this by installing `contextlib2`, which backports the Python 3 functionality to Python 2.)
2. `GeneratorContextManager` objects do not preserve the signature of the decorated functions. The decorated `hello` function above will have the generic signature `hello(*args, **kwargs)`, but fails if called with more than zero arguments.

For these reasons, the *decorator* module, starting from release 3.4, offers a `decorator.contextmanager` decorator that solves both problems, *and* works in all supported Python versions. Its usage is identical, and factories decorated with `decorator.contextmanager` will return instances of `ContextManager`, a subclass of the standard library's `contextlib.GeneratorContextManager` class. The subclass includes an improved `__call__` method, which acts as a signature-preserving decorator.

# The `FunctionMaker` class

You may wonder how the functionality of the `decorator` module is implemented. The basic building block is a `FunctionMaker` class. It generates on-the-fly functions with a given name and signature from a function template passed as a string.

If you're just writing ordinary decorators, then you probably won't need to use `FunctionMaker` directly. But in some circumstances, it can be handy. You will see an example shortly--in the implementation of a cool decorator utility (`decorator_apply`).

`FunctionMaker` provides the `.create` classmethod, which accepts the *name*, *signature*, and *body* of the function you want to generate, as well as the execution environment where the function is generated by `exec`.

Here's an example:

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

>>> f1 = FunctionMaker.create('f1(a, b)', 'f(a, b)', dict(f=f))
>>> f1(1,2)
(1, 2) {}
```

It is important to notice that the function body is interpolated before being executed; **be careful** with the `%` sign!

`FunctionMaker.create` also accepts keyword arguments. The keyword arguments are attached to the generated function. This is useful if you want to set some function attributes (e.g., the docstring `__doc__`).

For debugging/introspection purposes, it may be useful to see the source code of the generated function. To do this, just pass `addsource=True`, and the generated function will get a `__source__` attribute:

```
>>> f1 = FunctionMaker.create(
...     'f1(a, b)', 'f(a, b)', dict(f=f), addsource=True)
>>> print(f1.__source__)
def f1(a, b):
    f(a, b)
<BLANKLINE>
```

The first argument to `FunctionMaker.create` can be a string (as above), or a function. This is the most common usage, since you typically decorate pre-existing functions.

If you're writing a framework, however, you may want to use `FunctionMaker.create` directly, rather than `decorator`, because it gives you direct access to the body of the generated function.

For instance, suppose you want to instrument the `__init__` methods of a set of classes, by preserving their signature. (This use case is not made up. This is done by SQAlchemy, and other frameworks, too.) Here is what happens:

- If first argument of `FunctionMaker.create` is a function, an instance of `FunctionMaker` is created with the attributes `args`, `varargs`, `keywords`, and `defaults`. (These mirror the return values of the standard library's `inspect.getargspec`.)
- For each item in `args` (a list of strings of the names of all required arguments), an attribute `arg0`, `arg1`, ..., `argN` is also generated.
- Finally, there is a `signature` attribute, which is a string with the signature of the original function.

**NOTE:** You should not pass signature strings with default arguments (e.g., something like `'f1(a, b=None)'`). Just pass `'f1(a, b)'`, followed by a tuple of defaults:

```
>>> f1 = FunctionMaker.create(
...     'f1(a, b)', 'f(a, b)', dict(f=f), addsource=True, defaults=(None,))
>>> print(getargspec(f1))
ArgSpec(args=['a', 'b'], varargs=None, varkw=None, defaults=(None,))
```

# Getting the source code

Internally, `FunctionMaker.create` uses `exec` to generate the decorated function. Therefore `inspect.getsource` will not work for decorated functions. In IPython, this means that the usual `??` trick will give you the (right on the spot) message `Dynamically generated function. No source code available`.

In the past, I considered this acceptable, since `inspect.getsource` does not really work with "regular" decorators. In those cases, `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
```

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

```
>>> import inspect
>>> 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 still exists in all versions of Python, except Python 3.5.

However, there is a workaround. The decorated function has the `__wrapped__` attribute, pointing to the original function. The simplest way to get the source code is to call `inspect.getsource` on the undecorated function:

```
>>> print(inspect.getsource(factorial.__wrapped__))
@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 on the net you find some cool decorator that you would like to include in your code. However, more often than not, the cool decorator is not signature-preserving. What you need is 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 by preserving the signature even if dec
    is not a signature-preserving decorator.
```

```
    """
    return FunctionMaker.create(
        func, 'return decfunc(%(signature)s)',
        dict(decfunc=dec(func)), __wrapped__=func)
```

`decorator_apply` sets the generated function's `__wrapped__` attribute to the original function, so you can get the right source code. If you are using a Python later than 3.2, you should also set the `__qualname__` attribute to preserve the qualified name of the original function.

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

To give a good example for `decorator_apply`, I will show a pretty slick decorator that converts a tail-recursive function into an iterative function. I have shamelessly stolen the core concept 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
    with improvements by me and George Sakkis.
    """

    def __init__(self, func):
        self.func = func
        self.firstcall = True
        self.CONTINUE = object()  # sentinel

    def __call__(self, *args, **kwd):
        CONTINUE = self.CONTINUE
        if self.firstcall:
            func = self.func
            self.firstcall = False
            try:
                while True:
                    result = func(*args, **kwd)
                    if result is CONTINUE:  # update arguments
                        args, kwd = self.argskwd
                    else:  # last call
                        return result
            finally:
                self.firstcall = True
        else:  # return the arguments of the tail call
            self.argskwd = args, kwd
            return CONTINUE
```

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 thought! ;)

Notice that there is no recursion limit now; 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 does either of the following:

- returns a value without making a recursive call; or,
- returns directly the result of a recursive call.

# Python 3.5 coroutines

I am personally not using Python 3.5 coroutines yet, because at work we are still maintaining compatibility with Python 2.7. However, some users requested support for coroutines and since version 4.1 the decorator module has it. You should consider the support experimental and kindly report issues if you find any.

Here I will give a single example of usage. Suppose you want to log the moment a coroutine starts and the moment it stops for debugging purposes. You could write code like the following:

```
import time
import logging
from asyncio import get_event_loop, sleep, wait
from decorator import decorator

@decorator
async def log_start_stop(coro, *args, **kwargs):
    logging.info('Starting %s%s', coro.__name__, args)
    t0 = time.time()
    await coro(*args, **kwargs)
    dt = time.time() - t0
    logging.info('Ending %s%s after %d seconds', coro.__name__, args, dt)

@log_start_stop
async def make_task(n):
    for i in range(n):
        await sleep(1)
```

```
if __name__ == '__main__':
    logging.basicConfig(level=logging.INFO)
    tasks = [make_task(3), make_task(2), make_task(1)]
    get_event_loop().run_until_complete(wait(tasks))
```

and you will get at output like this:

```
INFO:root:Starting make_task(1,)
INFO:root:Starting make_task(3,)
INFO:root:Starting make_task(2,)
INFO:root:Ending make_task(1,) after 1 seconds
INFO:root:Ending make_task(2,) after 2 seconds
INFO:root:Ending make_task(3,) after 3 seconds
```

This may be handy if you have trouble understanding what it going on with a particularly complex chain of coroutines. With a single line you can decorate the troubling coroutine function, understand what happens, fix the issue and then remove the decorator (or keep it if continuous monitoring of the coroutines makes sense). Notice that *inspect.iscoroutinefunction(make_task)* will return then right answer (i.e. *True*).

# Multiple dispatch

There has been talk of implementing multiple dispatch functions (i.e. "generic functions") in Python for over ten years. Last year, something concrete was done for the first time. As of Python 3.4, we have the decorator `functools.singledispatch` to implement generic functions!

As its name implies, it is limited to *single dispatch*; in other words, it is able to dispatch on the first argument of the function only.

The `decorator` module provides the decorator factory `dispatch_on`, which can be used to implement generic functions dispatching on *any* argument. Moreover, it can manage dispatching on more than one argument. (And, of course, it is signature-preserving.)

Here is a concrete example (from a real-life use case) where it is desiderable to dispatch on the second argument.

Suppose you have an `XMLWriter` class, which is instantiated with some configuration parameters, and has the `.write` method which serializes objects to XML:

```
class XMLWriter(object):
    def __init__(self, **config):
        self.cfg = config

    @dispatch_on('obj')
    def write(self, obj):
        raise NotImplementedError(type(obj))
```

Here, you want to dispatch on the *second* argument; the first is already taken by `self`. The `dispatch_on` decorator factory allows you to specify the dispatch argument simply by passing its name as a string. (Note that if you misspell the name you will get an error.)

The decorated function *write* is turned into a generic function ( *write* is a function at the idea it is decorated; it will be turned into a method later, at class instantiation time), and it is called if there are no more specialized implementations.

Usually, default functions should raise a `NotImplementedError`, thus forcing people to register some implementation. You can perform the registration with a decorator:

```
@XMLWriter.write.register(float)
def writefloat(self, obj):
    return '<float>%s</float>' % obj
```

Now XMLWriter can serialize floats:

```
>>> writer = XMLWriter()
>>> writer.write(2.3)
'<float>2.3</float>'
```

I could give a down-to-earth example of situations in which it is desiderable to dispatch on more than one argument--for instance, I once implemented a database-access library where the first dispatching argument was the the database driver, and the second was the database record--but here I will follow tradition, and show the time-honored Rock-Paper-Scissors example:

```
class Rock(object):
    ordinal = 0
```

```
class Paper(object):
    ordinal = 1
```

```
class Scissors(object):
    ordinal = 2
```

I have added an ordinal to the Rock-Paper-Scissors classes to simplify the implementation. The idea is to define a generic function (`win(a, b)`) of two arguments corresponding to the *moves* of the first and second players. The *moves* are instances of the classes Rock, Paper, and Scissors.

Paper wins over Rock; Scissors wins over Paper; and Rock wins over Scissors.

The function will return +1 for a win, -1 for a loss, and 0 for parity. There are 9 combinations, but combinations with the same ordinal (i.e. the same class) return 0. Moreover, by exchanging the order of the arguments, the sign of the result changes. Therefore, it is sufficient to directly specify only 3 implementations:

```
@dispatch_on('a', 'b')
def win(a, b):
    if a.ordinal == b.ordinal:
        return 0
    elif a.ordinal > b.ordinal:
        return -win(b, a)
    raise NotImplementedError((type(a), type(b)))
```

```
@win.register(Rock, Paper)
def winRockPaper(a, b):
    return -1
```

```
@win.register(Paper, Scissors)
def winPaperScissors(a, b):
    return -1
```

```
@win.register(Rock, Scissors)
def winRockScissors(a, b):
    return 1
```

Here is the result:

```
>>> win(Paper(), Rock())
1
>>> win(Scissors(), Paper())
1
>>> win(Rock(), Scissors())
1
>>> win(Paper(), Paper())
0
>>> win(Rock(), Rock())
0
>>> win(Scissors(), Scissors())
0
>>> win(Rock(), Paper())
-1
>>> win(Paper(), Scissors())
-1
>>> win(Scissors(), Rock())
-1
```

The point of generic functions is that they play well with subclassing. For instance, suppose we define a `StrongRock`, which does not lose against Paper:

```
class StrongRock(Rock):
    pass
```

```
@win.register(StrongRock, Paper)
def winStrongRockPaper(a, b):
    return 0
```

Then you do not need to define other implementations; they are inherited from the parent:

```
>>> win(StrongRock(), Scissors())
1
```

You can introspect the precedence used by the dispath algorithm by calling `.dispatch_info(*types)`:

```
>>> win.dispatch_info(StrongRock, Scissors)
[('StrongRock', 'Scissors'), ('Rock', 'Scissors')]
```

Since there is no direct implementation for (`StrongRock`, `Scissors`), the dispatcher will look at the implementation for (`Rock`, `Scissors`) which is available. Internally, the algorithm is doing a cross product of the class precedence lists (or *Method Resolution Orders*, MRO for short) of `StrongRock` and `Scissors`, respectively.

# Generic functions and virtual ancestors

In Python, generic functions are complicated by the existence of "virtual ancestors": superclasses which are not in the class hierarchy.

Consider this class:

```
class WithLength(object):
    def __len__(self):
        return 0
```

This class defines a `__len__` method, and is therefore considered to be a subclass of the abstract base class `collections.Sized`:

```
>>> issubclass(WithLength, collections.Sized)
True
```

However, `collections.Sized` is not in the MRO of `WithLength`; it is not a true ancestor. Any implementation of generic functions (even with single dispatch) must go through some contorsion to take into account the virtual ancestors.

In particular, if we define a generic function...

```
@dispatch_on('obj')
def get_length(obj):
    raise NotImplementedError(type(obj))
```

...implemented on all classes with a length...

```
@get_length.register(collections.Sized)
def get_length_sized(obj):
    return len(obj)
```

...then `get_length` must be defined on `WithLength` instances...

```
>>> get_length(WithLength())
0
```

...even if `collections.Sized` is not a true ancestor of `WithLength`.

Of course, this is a contrived example--you could just use the builtin `len`--but you should get the idea.

Since in Python it is possible to consider any instance of `ABCMeta` as a virtual ancestor of any other class (it is enough to register it as `ancestor.register(cls)`), any implementation of generic functions must be aware of the registration mechanism.

For example, suppose you are using a third-party set-like class, like the following:

```
class SomeSet(collections.Sized):
    # methods that make SomeSet set-like
    # not shown ...
    def __len__(self):
        return 0
```

Here, the author of `SomeSet` made a mistake by inheriting from `collections.Sized` (instead of `collections.Set`).

This is not a problem. You can register *a posteriori* `collections.Set` as a virtual ancestor of `SomeSet`:

```
>>> _ = collections.Set.register(SomeSet)
>>> issubclass(SomeSet, collections.Set)
True
```

Now, let's define an implementation of `get_length` specific to set:

```
@get_length.register(collections.Set)
def get_length_set(obj):
    return 1
```

The current implementation (and `functools.singledispatch` too) is able to discern that a `Set` is a `Sized` object, by looking at the class registry, so it uses the more specific implementation for `Set`:

```
>>> get_length(SomeSet())  # NB: the implementation for Sized would give 0
1
```

Sometimes it is not clear how to dispatch. For instance, consider a class `C` registered both as `collections.Iterable` and `collections.Sized`, and defines a generic function `g` with implementations for both `collections.Iterable` *and* `collections.Sized`:

```
def singledispatch_example1():
    singledispatch = dispatch_on('obj')

    @singledispatch
    def g(obj):
        raise NotImplementedError(type(g))

    @g.register(collections.Sized)
    def g_sized(object):
        return "sized"

    @g.register(collections.Iterable)
    def g_iterable(object):
        return "iterable"

    g(C())  # RuntimeError: Ambiguous dispatch: Iterable or Sized?
```

It is impossible to decide which implementation to use, since the ancestors are independent. The following function will raise a `RuntimeError` when called. This is consistent with the "refuse the temptation to guess" philosophy. `functools.singledispatch` would raise a similar error.

It would be easy to rely on the order of registration to decide the precedence order. This is reasonable, but also fragile:

- if, during some refactoring, you change the registration order by mistake, a different implementation could be taken;
- if implementations of the generic functions are distributed across modules, and you change the import order, a different implementation could be taken.

So the `decorator` module prefers to raise an error in the face of ambiguity. This is the same approach taken by the standard library.

However, it should be noted that the *dispatch algorithm* used by the decorator module is different from the one used by the standard library, so in certain cases you will get different answers. The difference is that

`functools.singledispatch` tries to insert the virtual ancestors *before* the base classes, whereas `decorator.dispatch_on` tries to insert them *after* the base classes.

Here's an example that shows the difference:

```
def singledispatch_example2():
    # adapted from functools.singledispatch test case
    singledispatch = dispatch_on('arg')

    class S(object):
        pass

    class V(c.Sized, S):
        def __len__(self):
            return 0

    @singledispatch
    def g(arg):
        return "base"

    @g.register(S)
    def g_s(arg):
        return "s"

    @g.register(c.Container)
    def g_container(arg):
        return "container"

    v = V()
    assert g(v) == "s"
    c.Container.register(V)  # add c.Container to the virtual mro of V
    assert g(v) == "s"  # since the virtual mro is V, Sized, S, Container
    return g, V
```

If you play with this example and replace the `singledispatch` definition with `functools.singledispatch`, the assertion will break: `g` will return `"container"` instead of `"s"`, because `functools.singledispatch` will insert the `Container` class right before `S`.

Notice that here I am not making any bold claim such as "the standard library algorithm is wrong and my algorithm is right" or viceversa. It just point out that there are some subtle differences. The only way to understand what is really happening here is to scratch your head by looking at the implementations. I will just notice that `.dispatch_info` is quite essential to see the class precedence list used by algorithm:

```
>>> g, V = singledispatch_example2()
>>> g.dispatch_info(V)
[('V',), ('Sized',), ('S',), ('Container',)]
```

The current implementation does not implement any kind of cooperation between implementations. In other words, nothing is akin either to call-next-method in Lisp, or to `super` in Python.

Finally, let me notice that the decorator module implementation does not use any cache, whereas the `singledispatch` implementation does.

# Caveats and limitations

One thing you should be aware of, is the performance penalty of decorators. The worse case is shown by the following example:

```
$ cat performance.sh
python3 -m timeit -s "
from decorator import decorator

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

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

python3 -m timeit -s "
def f():
    pass
" "f()"
```

On my laptop, using the `do_nothing` decorator instead of the plain function is five times slower:

```
$ bash performance.sh
1000000 loops, best of 3: 1.39 usec per loop
1000000 loops, best of 3: 0.278 usec per loop
```

Of course, a real life function probably does something more useful than the function `f` here, so the real life performance penalty *could* be negligible. As always, the only way to know if there is a penalty in your specific use case is to measure it.

More importantly, 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()` gives you a `ZeroDivisionError`. But since the function is decorated, the traceback is longer:

```
>>> f()
Traceback (most recent call last):
  ...
    File "<string>", line 2, in f
    File "<doctest __main__[22]>", line 4, in trace
      return f(*args, **kw)
    File "<doctest __main__[51]>", line 3, in f
      1/0
ZeroDivisionError: ...
```

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* responsible for the performance penalty, since is the called *only once* (at function decoration time); it is *not* called each time the decorated function is called.

Presently, there is no clean way to avoid `exec`. A clean solution would require changing the CPython implementation, by adding a hook to functions (to allow changing their signature directly).

Even in Python 3.5, it is impossible to change the function signature directly. Thus, the `decorator` module is still useful! As a matter of fact, this is the main reason why I still maintain the module and release new versions.

It should be noted that in Python 3.5, a *lot* of improvements have been made: you can decorate a function with `func_tools.update_wrapper`, and `pydoc` will see the correct signature. Unfortunately, the function will still have an incorrect signature internally, as you can see by using `inspect.getfullargspec`; so, all documentation tools using `inspect.getfullargspec` - which has been rightly deprecated - will see the wrong signature.

In the present implementation, decorators generated by `decorator` can only be used on user-defined Python functions or methods. They cannot be used on generic callable objects or built-in functions, due to limitations of the standard library's `inspect` module, especially for Python 2. In Python 3.5, many such limitations have been removed, but I still think that it is cleaner and safer to decorate only functions. If you want to decorate things like classmethods/staticmethods and general callables - which I will never support in the decorator module - I suggest you to look at the wrapt project by Graeme Dumpleton.

There is a strange quirk when decorating functions with keyword arguments, if one of the arguments has the same name used in the caller function for the first argument. The quirk was reported by David Goldstein.

Here is an example where it is manifest:

```
>>> @memoize
... def getkeys(**kw):
...     return kw.keys()
>>> getkeys(func='a')
Traceback (most recent call last):
 ...
TypeError: _memoize() got multiple values for ... 'func'
```

The error message looks really strange... until you realize that the caller function *_memoize* uses *func* as first argument, so there is a confusion between the positional argument and the keywork arguments.

The solution is to change the name of the first argument in *_memoize*, or to change the implementation like so:

```
def _memoize(*all_args, **kw):
    func = all_args[0]
    args = all_args[1:]
    if kw:  # frozenset is used to ensure hashability
        key = args, frozenset(kw.items())
    else:
        key = args
    cache = func.cache  # attribute added by memoize
    if key not in cache:
        cache[key] = func(*args, **kw)
    return cache[key]
```

This avoids the need to name the first argument, so the problem simply disappears. This is a technique that you should keep in mind when writing decorators for functions with keyword arguments. Also, notice that lately I have come to believe that decorating functions with keyword arguments is not such a good idea, and you may want not to do that.

On a similar note, there is a restriction on argument names. For instance, if you name 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 makes a (shallow) copy of the original function dictionary:

```
>>> 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'
```

# LICENSE (2-clause BSD)



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!