The `decorator` module

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

Version: 4.0.1 (2015-09-25)
Supports: Python 2.6, 2.7, 3.0, 3.1, 3.2, 3.3, 3.4, 3.5
Download page: http://pypi.python.org/pypi/decorator/4.0.1
Installation: `pip install decorator`

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. This version supports all Python releases from 2.6 up to 3.5, which currently is still in beta status.

What's new

Since now there is a single manual for all Python versions, I took the occasion for overhauling the documentation. Therefore, even if you are an old time user, you may want to read the docs again, since several examples have been improved. The packaging has been improved and I am distributing the code in wheel format too. The integration with setuptools has been improved and now you can use `python setup.py test` to run the tests. A new utility function `decorate(func, caller)` has been added, doing the same job that in the past was done by `decorator(caller, func)`. The old functionality is still there for compatibility sake, but it is deprecated and not documented anymore.

Apart from that, there is a new experimental feature. The decorator module now includes an implementation of generic (multiple dispatch) functions. The API is designed to mimic the one of `functools.singledispatch` (introduced in Python 3.4) but the implementation is much simpler; moreover all the decorators involved preserve the signature of the decorated functions. For the moment the facility is there mostly to exemplify the power of the module. In the future it could be enhanced/optimized; on the other hand, both its behavior and its API could change. Such is the fate of experimental features. In any case it is very short and compact (less then one hundred lines) so you can extract it for your own use. Take it as food for thought.

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 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 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, 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. In recent versions of Python you can find a sophisticated `lru_cache` decorator in the standard library (in `functools`). Here I am just interested in giving an example.

A simple implementation could be the following (notice that in general it is impossible to memoize correctly 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 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). Here is an example of usage:

```
@memoize_uw
def f1(x):
    "Simulate some long computation"
```

```
    time.sleep(1)
    return x
```

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 `memoize_uw` returns a function with a *different signature* from the original function.

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 such as `pydoc` will give wrong informations about the signature of `f1`, unless you are using Python 3.5. 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 positional argument (2 given)
```

Notice even in Python 3.5 `inspect.getargspec` and `inspect.getfullargspec` (which are deprecated in that release) will give the wrong signature.

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, 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.items())
    else:
        key = args
    cache = func.cache  # attribute added by memoize
```

```
    if key not in cache:
        cache[key] = func(*args, **kw)
    return cache[key]
```

At this point 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 responsability to clear it, if needed.
    """
    f.cache = {}
    return decorate(f, _memoize)
```

The difference with respect to the `memoize_uw` approach, which is based on nested functions, is that the decorator module forces you to lift the inner function at the outer level. Moreover, you are forced to pass explicitly 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

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

Function annotations

Python 3 introduced the concept of function annotations,i.e. 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, is able to understand and to preserve the 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`, new in Python 3 (and deprecated in favor of `inspect.signature` in Python 3.5):

```
>>> 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. Such an attribute is added to be consistent with the way `functools.update_wrapper` work. Another attribute which is copied from the original function is `__qualname__`, the qualified name. This is an attribute which is present starting from Python 3.3.

decorator.decorator

It may be annoying to write a caller function (like the `_trace` function above) and then a trivial wrapper (`def trace(f): return decorate(f, _trace)`) every time. For this reason, the `decorator` module provides an easy shortcut to convert the caller function into a signature-preserving decorator: 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 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):
...     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 now `trace` will be 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`, and sometimes it is best to have back a "busy" message than to block everything. This behavior can be implemented 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 of instances of that class.

As an example, here will 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. This is very similar to the approach used in the `concurrent.futures` package. Of course the code here is just an example, it is not a recommended way of implementing futures. The implementation is the following:

```
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, which has a `.result()` method which blocks until the underlying thread finishes and returns the final result. Here is a minimalistic example of usage:

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

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

contextmanager

For a long time Python had in its standard library a `contextmanager` decorator, able to convert generator functions into `GeneratorContextManager` factories. For instance if you write

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

then `before_after` is a factory function returning `GeneratorContextManager` objects which can be used with 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 as in this example:

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

The `ba` decorator is basically inserting a `with ba:` block inside the function. However there two issues: the first is that `GeneratorContextManager` objects are callable only in Python 3.2, so the previous example will break in older versions of Python (you can solve this by installing `contextlib2`); the second is that `GeneratorContextManager` objects do not preserve the signature of the decorated functions: the decorated `hello` function here will have a generic signature `hello(*args, **kwargs)` but will break when called with more than zero arguments. For such reasons the decorator module, starting with release 3.4, offers a `decorator.contextmanager` decorator that solves both problems and works in all supported Python versions. The usage is the same and factories decorated with `decorator.contextmanager` will returns instances of `ContextManager`, a subclass of `contextlib.GeneratorContextManager` with a `__call__` method acting as a signature-preserving decorator.

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. You will see an example shortly, in the implementation of a cool decorator utility (`decorator_apply`).

`FunctionMaker` provides a `.create` classmethod which takes as input the name, signature, and body of the function we want to generate as well as the execution environment were the function is generated by `exec`. Here is 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, so be careful with the `%` sign!

`FunctionMaker.create` 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 generated function:

```
>>> 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>
```

`FunctionMaker.create` can take as first argument a string, as in the examples before, or a function. This is the most common usage, since typically you want to decorate a pre-existing function. A framework author may want to use directly `FunctionMaker.create` instead of `decorator`, since 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 (such use case is not made up; this is done in SQAlchemy and in other frameworks). When the first argument of `FunctionMaker.create` is a function, a `FunctionMaker` object is instantiated internally, with attributes `args`, `varargs`, `keywords` and `defaults` which are the the return values of the standard library function `inspect.getargspec`. For each argument in the `args` (which is a list of strings containing the names of the mandatory arguments) an attribute `arg0`, `arg1`, ..., `argN` is also generated. Finally, there is a `signature` attribute, a string with the signature of the original function.

Notice: you should not pass signature strings with default arguments, i.e. something like `'f1(a, b=None)'`. Just pass `'f1(a, b)'` and then 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. 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
```

```
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 is still there, in all versions of Python except Python 3.5, which is not yet released. There is however a workaround. The decorated function has an attribute `__wrapped__`, pointing 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.__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 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 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 attribute `__wrapped__` of the generated function to the original function, so that you can get the right source code. If you are using a Python more recent 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 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
    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 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).

Multiple dispatch

There has been talk of implementing multiple dispatch (i.e. generic) functions in Python for over ten years. Last year for the first time something concrete was done and now in Python 3.4 we have a decorator `functools.singledispatch` which can be used to implement generic functions. As the name implies, it has the restriction of being limited to single dispatch, i.e. it is able to dispatch on the first argument of the function only. The decorator module provide a 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 I will give a very concrete example (taken 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 a `.write` method which is able to serialize 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 since the first, `self` is already taken. The `dispatch_on` decorator factory allows you to specify the dispatch argument by simply passing its name as a string (notice that if you mispell the name you will get an error). The function decorated is turned into a generic function and it is the one which is called if there are no more specialized implementations. Usually such default function should raise a `NotImplementedError`, thus forcing people to register some implementation. The registration can be done with a decorator:

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

Now the XMLWriter is able to 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 once I implemented a database-access library where the first dispatching argument was the the database driver and the second one was the database record), but here I prefer to follow the 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 player respectively. 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, however 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, so it is enough to specify directly 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 we do not need to define other implementations, since 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

Generic function implementations in Python are complicated by the existence of "virtual ancestors", i.e. superclasses which are not in the class hierarchy. Consider for instance this class:

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

This class defines a `__len__` method and as such is 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 since 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 take virtual ancestors into account. Let me give an 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 not inheriting from `collections.Set`, but only from `collections.Sized`.

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

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

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

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

The current implementation, as the one used by `functools.singledispatch`, is able to discern that a `Set` is a `Sized` object, so the more specific implementation for `Set` is taken:

```
>>> 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 define a generic function `g` with implementations both for `collections.Iterable` and `collections.Sized`. It is impossible to decide which implementation to use, since the ancestors are independent, and the following function will raise a RuntimeError when called:

```
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?
```

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 noticed that the dispatch algorithm used by the decorator module is different from the one used by the standard library, so there are cases where 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. I will give an example showing 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 assert will break: `g` will return `"container"` instead of `"s"`, because `functools.singledispatch` will insert the `Container` class right before `S`. The only way to understand what is happening here is to scratch your head by looking at the implementations. I will just notice that `.dispatch_info` is quite useful:

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

The current implementation does not implement any kind of cooperation between implementations, i.e. there is nothing akin to call-next-method in Lisp, nor akin to `super` in Python.

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

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
```

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.

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()` 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__[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* 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. However, at present, even in Python 3.5 it is impossible to change the function signature directly, therefore the `decorator` module is still useful. Actually, this is the main reasons why I keep maintaining the module and releasing new versions. It should be noticed that in Python 3.5 a lot of improvements have been made: in that version you can decorated a function with `func_tools.update_wrapper` and `pydoc` will see the correct signature; still internally the function will have an incorrect signature, as you can see by using `inspect.getfullargspec`: all documentation tools using such function (which has been correctly 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, not on generic callable objects, nor on built-in functions, due to limitations of the `inspect` module in the standard library, especially for Python 2.X (in Python 3.5 a lot of such limitations have been removed).

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 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

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!