dectools Documentation

Release 0.1.4

Charles Merriam

March 17, 2010

CONTENTS

aw.. dectools documentation on Thu Mar 4 16:11:14 2010.

Contents:

CHAPTER

ONE

INTRODUCTION

The dectools module overcomes some challenges in the Python decorators. It provides a clear and convenient method for writing your own decorators. The dectools modules also provides a library of common decorations for logging and testing your code.

The prebuilt libraries can be used with minimal understanding and provide immediate benefit to users. The constructors for writing your own decorators require following a convention for naming the required arguments of your decorator. In all cases, care is taken to provide correct function signatures, metadata, and record when decorators are applied in the function metadata.

The support for class decoration is still coming.

1.1 Prebuilt Decorators (the library)

Prebuilt decorators can be of immediate benefit to users and can be used without a good understanding of decorators. These are used for common problems such as caching, logging, and programming with invariants or pre-conditions and post-conditions.

1.2 Make_call Decorators (the constructors)

There are five special constructors that allow users to make their own decorators that are called before, after, if (conditionally), instead, or once during declaration of another function. These provide for most use cases of writing custom decorators. Using these constructors removes the boilerplate or copy-and-paste code associated with writing your own decorators, while also handling the function signatures, the metadata, and a history of which functions decorate others.

1.3 I Don't Read Documentation

Typical usage looks like this:

```
from dectools import *   # <-- Don't do this

@make_call_once
def register_callback(function):
      gui.callback_create(function.__name__, function)

@make_call_before
def require_login(function, args, kwargs, page_name):
```

```
    while not current_user_id():
        ...

@log()
@register()
@require_login("Summary of Items")
def view_summary():
    ...
```

1.4 Thanks

Thanks to Michele Simionato for exposing utility functions in Decorator Decorator. Thanks to David Mertz for his excellent IBM Developer Works article "Charming Python: Decorators make magic easy", http://www.ibm.com/developerworks/linux/library/l-cpdecor.html

1.5 Supported Versions

Currently, only version Python 2.6.x is guaranteed to work.

1.6 Contact

You can contact the author, Charles Merriam, at charles.merriam@gmail.com.

1.7 Sources

- The source and bugs repository is git://github.com/merriam/dectools
- The package is also available from the cheeseshop, http://pypi.python.org/pypi/dectools
- The presentation from PyCon 2010, of decorators and introducing the library, is in the docs/ directory.
- The video from the PyCon 2010 presentation is available at http://blip.tv/file/3257278/

CHAPTER

TWO

CHANGES BETWEEN VERSIONS

2.1 Version Plan

Expect that the minor version number will change rapidly, usually whenever minor API changes are introduced. I do not expect to keep the version number under 1.0 for six years: I never understood that trend. Until I get significant feedback on the module, expect the API to change randomly.

2.2 Version History

This echoes the file "CHANGES.txt"

```
v0.1.0, 18 Feb 2010 -- Initial escape into the public.
v0.1.1, 20 Feb 2010 -- Apparently distutils included no files.   Added
                       slides in the documentation directory.  Started
                       a public repository at
                       git://github.com/merriam/dectools.git
v0.1.2, 20 Feb 2010 -- Making it work a bit more.  Now need to
                       include dectools.dectools as dectools in the tests.
v0.1.3, 27 Feb 2010 -- Tests work as both nosetests and Unittest.  94% coverage.
                       Started a manual in docs/manual.txt.
v0.1.4, 17 Mar 2010 -- Documentation and packaging update.
                       Include a Sphinx manual.  Add a get_version() function.  Add
                       a make_pkg script.
```

CHAPTER

THREE

USING THE PREBUILT DECORATORS

To use the library decorators, add this import line:

```
import dectools
```

3.1 Debugging Decorators

3.1.1 @log()

Print a log all parameters passed to the function, and the return value or exception.

You can use the log decorator to print a message before and after each function call. The basic output is one statement to standard output when your function is called and one when your function returns. There are also parameters to change the output destination and a parameter to each before and after calling that take functions. I may change this API slightly to take a format string for before and after functions.

Recipe: Basic Usage

Code:

```
from dectools import log

@log
def add_two(first, second):
    return first + second

total = add_two(5, second=3)
```

and *Output*:

```
>add_two(first=5, second=3)
<add_two. return value: 8
```

You can find additional examples in the test case: *test/test_log.py*

3.1.2 @pre and @post

Assert an expression is true before or after the function is executed.

These decorators take an expression and execute either before, for @pre, or after, for @post, your function. You can use both @pre and @post to decorate a single function. Internally, this uses the Python eval() function, which creates an issue for locals and globals.

If your expressions check only the parameters or variables referenced through the self parameter, you can use just function like:

```
from dectools import pre, post

@post('self.total >= 0 and self.tax >= 0')
@pre('item and item.name and item.price >= 0')
def add_to_purchase(self, item):
    do_something()
```

If you have a pre or post condition that references other global or local variable, you should pass the current globals and locals as parameters, as in this example:

```
@post('Database.get_item_by_name(name) > 0', globals(), locals())
def add_new_item_to_database(name, description, price):
    pass
```

3.1.3 @invariant

A class decorator to force calling the function self._invariant() after the __init__() method and before and after each public method. A public method of a class is one that is not a @classmethod or @staticmethod and does not start with an underscore. A public method takes 'self' as its first parameter.

CHAPTER

FOUR

USING THE MAKE_CALL_* CONSTRUCTORS

To use the library constructors, add this import line:

```
import dectools
```

Use a constructor as decorator on your function. Your function must have the correct function signature: the first argument must be "function" and the second and third arguments are probably "args" and "kwargs". Put any additional arguments after these required arguments.

TODO: Make these arguments optional, because only @make_call_instead really needs them.

For example:

```
@make_call_before
def my_new_call_before_decorator(function, args, kwargs, some_other_argument):
     # do stuff
     ....

@my_new_call_before_decorator("foo")  # or (some_other_argument = "foo")
def my_newly_decorated_function(other_args):
    pass
```

We will call "my_new_call_before_decorator" the decorator and "my_newly_decorated_function" the decorated function.

4.1 Quick Reference

Construtor	Required Parameters	Used For
@make_call_once	function	registering with frameworks, global resource creation
@make_call_before	funciton, args, kwargs	requiring login, lazy instantiation
@make_call_if	function, args, kwargs	checking priviledges, skipping if overloaded
@make_call_after	function, args, kwargs	converting return values to exceptions, redrawing the screen
@make_call_instead	function, args, kwargs	everything else. logs, caches, locking, resource handling.

4.2 @make_call_once

Requires: `(function, ...)`

Call the decorator once when the decorator is applied, but does not affect the decorated function.

Used for registering the function with frameworks, global resource creation, or enforcing proprietary licensing restrictions.

4.3 @make_call_before

Requires: `(function, args, kwargs, ...)`

Call the decorator before the decorated function, then call the decorated function. If the decorator throws an exception, then the decorated function is not called.

Used for requiring users to be logged in first, lazy instantiation of resources, and setting preconditions.

4.4 @make_call_after

Requires: `(function, args, kwargs, ...)`

Call decorator after the decorated function. The decorator's return value is returned.

Used for converting errors from return values into exceptions, double checking that resources were released, and redrawing of the screen.

4.5 @make_call_if

Requires: `(function, args, kwargs, ...)`

Call decorator first. If decorator returns a True value, then call the decorated function.

Used for checking authorization, skipping decluttering during peak times, and enforcing some proprietary licensing.

4.6 @make_call_instead

Requires: `(function, args, kwargs, ...)`

Call the decorator. Do not call the decorated function. The decorator will usually call the function with the statement `return_value = function(*args, **kwargs)`.

Used for everything, including logging, caching, acquiring and releasing locks, acquiring and releasing other resources, comparing results between an optimized algorithm and a slow but reliable one, and more.

CHAPTER

FIVE

HOW THE CONSTRUCTORS REALLY WORK

TODO: Make the constructors really work this way instead of with a couple extra levels of functino calls. The concept is about right.

This follows through one example to show how it works. Walking through it might demystify decorators, might make you think I am an idiot, and might put you to sleep.

5.1 The example

```
@make_call_instead   # Step 1
def notice_me(function, args, kwargs, message = "I see you"):
    print message + ": " + function.__name__
    return function(args, kwargs)

@notice_me("Watching you")  # Step 2, Parts A and B.
def hello(name):
    print "Hello", name

hello("Charles")  # Step 3.
```

5.2 The example (unwrapped)

This is the same example, except we do some extra assignements to temporary or differently-named variables. It also moves the `@` operation to after the `def` statement, which is where it actually occurs.

```
def notice_me(function, args, kwargs, message = "I see you"):
    print message + ": " + function.__name__
    return function(args, kwargs)
widget = make_call_instead(notice_me)  # Step 1
notice_me = widget


def hello(name):
    print "Hello", name
gizmo = widget("Watching you")  # Step 2 Part A
hello_out = gizmo(hello)        # Step 2 Part B
```

```
hello = hello_out

hello("Charles")                # Step 3
```

5.3 The explanation

5.3.1 Step 1

The constructor, @make_call_instead, is a concrete decorator because it takes one function as input. The function passed in, e.g., notice_me():

- has "function, args, and kwargs" as first three arguments. In the current version, this is asserted. In future versions, it might not be required.
- has zero or more additional arguments. The additional arguments may or may not have default values.

The return value, named widget here, has some properties:

- It has a signature of only the additional parameters. For this case, it has a signature of taking only the message argument.
- It has the __name__ metadata of notice_me().
- It has decorator data describing that "the function signature was changed" and mentions "make_call_instead()" as the culprit.
- It can be used in step 2.

5.3.2 Step 2, Part A

Step 2 is divided into two parts, because two separate operations occur in the line @notice_me("Watching you"). notice_me("Watching you") resolves into an intermediate value and the '@' operation is then applied.

For Part A., gizmo = widget("Watching you") takes an argument.

- The argument corresponds to the additional arguments after "function, args, kwargs".
- The signature of widget is "def widget(message):" and widget("watching you") is equivalent to `widget(message = "watching you")`. Your IDE will help get the arguments correct.
- Widget is a complex decorator, taking arguments and yielding a decorator.
- dectools makes complex decorators out of functions. "Function" is used interchangably with "method".
- The output, gizmo(), is a concrete decorator for use in Step 2, Part B. gizmo() takes one argument, a callable function, and returns a callable function with the additional functionality of notice_me(). This makes gizmo() a concrete decorator.
- When you use the decorator again, like *@notice_me("Still watching")*, or, equivalently, *gizmo_2 = widget("Still_watching")* both gizmo and gizmo_2 will have the same __code__ attribute but different __closure__ attributes. If that made no sense, don't worry. There will not be a quiz. Just recognize that you will use notice_me with different input parameters.
- the output, gizmo, is used in Step 2, Part B.

5.3.3 Step 2, Part B

This line, "hello_out = gizmo(hello)" decorates hello.

- gizmo() takes exactly one argument, hello, and returns the modified function. For the example, we call it hello_out for clarity. Right after this line, hello_out is assigned to hello.
- hello_out has the same signature and `__name__` as hello. These are not related to the signature or name of notice_me or gizmo. That is, using the decorator does not change the signature: one of the magic points of dectools.
- hello is expected to be a function. The library might support hello being a class in the future if I can make a case for it being helpful. It works and is helpful for @make_call_once().
- hello_out does get additional metadata to record that it was decorated.

5.3.4 Step 3

When the line hello("charles") is called these things happen:

1. hello("charles") is called. When it was decorated, the actual code for hello was changed to call a dectools function. Let's call this function "call_through()" for this example.
2. "call_through()" then calls notice_me() with the original hello function and additonal decorator parameters. In this case, it's called:

   ```
   notice_me(function= hello, args = ("charles"), kwargs = {}, message = "Watching you")
   ```

3. notice_me() then prints "Watching you: hello", then calls hello. I wonder if anyone reads this documentation? Email me if you do.
4. The return value of hello() is returned to notice_me() which is returned to call_through() which is returned to the main program, where it is ignored.

CHAPTER

SIX

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*