pydecor Documentation

Release 2.0.0

Matthew Planchard

Jan 13, 2020

Contents

CHAPTER 1

pydecor package

1.1 Subpackages

1.1.1 pydecor.decorators package

Submodules

pydecor.decorators.generic module

pydecor.decorators.ready_to_wear module

Module contents

1.2 Submodules

1.2.1 pydecor.caches module

1.2.2 pydecor.constants module

1.2.3 pydecor.functions module

1.3 Module contents

CHAPTER 2

PyDecor

Easy-peasy Python decorators!

- GitHub: https://github.com/mplanchard/pydecor
- PyPI: https://pypi.python.org/pypi/pydecor
- Docs: https://pydecor.readthedocs.io/
- Contact: `msplanchard @ gmail` or @msplanchard on Twitter

2.1 Summary

Decorators are great, but they're hard to write, especially if you want to include arguments to your decorators, or use your decorators on class methods as well as functions. I know that, no matter how many I write, I still find myself looking up the syntax every time. And that's just for simple function decorators. Getting decorators to work consistently at the class and method level is a whole 'nother barrel of worms.

PyDecor aims to make function decoration easy and straightforward, so that developers can stop worrying about closures and syntax in triply nested functions and instead get down to decorating!

Table of Contents

2.2 Quickstart

Install `pydecor`:

```
pip install pydecor
```

Use one of the ready-to-wear decorators:

```
# Memoize a function

from pydecor import memoize


@memoize()
def fibonacci(n):
    """Compute the given number of the fibonacci sequence"""
    if n < 2:
        return n
    return fibonacci(n - 2) + fibonacci(n - 1)

print(fibonacci(150))
```

```
# Intercept an error and raise a different one

from flask import Flask
from pydecor import intercept
from werkzeug.exceptions import InternalServerError


app = Flask(__name__)


@app.route('/')
@intercept(catch=Exception, reraise=InternalServerError,
           err_msg='The server encountered an error rendering "some_view"')
def some_view():
    """The root view"""
    assert False
    return 'Asserted False successfully!'


client = app.test_client()
response = client.get('/')
```

(continues on next page)

(continued from previous page)

```
assert response.status_code == 500
assert 'some_view'.encode() in resp.data
```

Use a generic decorator to run your own functions `@before`, `@after`, or `@instead` of another function, like in the following example, which sets a User-Agent header on a Flask response:

```
from flask import Flask, make_response
from pydecor import Decorated, after


app = Flask(__name__)

# `Decorated` instances are passed to your functions and contain
# information about the wrapped function, including its `args`,
# `kwargs`, and `result`, if it's been called.

def set_user_agent(decorated: Decorated):
    """Sets the user-agent header on a result from a view"""
    resp = make_response(decorated.result)
    resp.headers.set('User-Agent', 'my_applicatoin')
    return resp


@app.route('/')
@after(set_user_agent)
def index_view():
    return 'Hello, world!'


client = app.test_client()
response = client.get('/')
assert response.headers.get('User-Agent') == 'my_application'
```

Or make your own decorator with `construct_decorator`

```
from flask import reques
from pydecor import Decorated, construct_decorator
from werkzeug.exceptions import Unauthorized


def check_auth(_decorated: Decorated, request):
    """Theoretically checks auth.

    It goes without saying, but this is example code. You should
    not actually check auth this way!
    """
    if request.host != 'localhost':
        raise Unauthorized('locals only!')


authed = construct_decorator(before=check_auth)


app = Flask(__name__)
```

(continues on next page)

(continued from previous page)

```
@app.route('/')
# Any keyword arguments provided to any of the generic decorators are
# passed directly to your callable.
@authed(request=request)
def some_view():
    """An authenticated view"""
    return 'This is sensitive data!'
```

2.3 Why PyDecor?

- **It's easy!**

  With PyDecor, you can go from this:

```
from functools import wraps
from flask import request
from werkzeug.exceptions import Unauthorized
from my_pkg.auth import authorize_request

def auth_decorator(request=None):
    """Check the passed request for authentication"""

    def decorator(decorated):

        @wraps(decorated)
        def wrapper(*args, **kwargs):
            if not authorize_request(request):
              raise Unauthorized('Not authorized!')
            return decorated(*args, **kwargs)
        return wrapper

    return decorated

@auth_decorator(request=requst)
def some_view():
    return 'Hello, World!'
```

  to this:

```
from flask import request
from pydecor import before
from werkzeug.exceptions import Unauthorized
from my_pkg.auth import authorize_request

def check_auth(_decorated, request=request):
    """Ensure the request is authorized"""
    if not authorize_request(request):
      raise Unauthorized('Not authorized!')

@before(check_auth, request=request)
def some_view():
    return 'Hello, world!'
```

  Not only is it less code, but you don't have to remember decorator syntax or mess with nested functions. Full disclosure, I had to look up a decorator sample to be sure I got the first example's syntax right, and I just spent

two weeks writing a decorator library.

- **It's fast!**

  PyDecor aims to make your life easier, not slower. The decoration machinery is designed to be as efficient as is reasonable, and contributions to speed things up are always welcome.

- **Implicit Method Decoration!**

  Getting a decorator to "roll down" to methods when applied to a class is a complicated business, but all of PyDecor's decorators provide it for free, so rather than writing:

```
from pydecor import log_call

class FullyLoggedClass(object):

    @log_call(level='debug')
    def some_function(self, *args, **kwargs):
        return args, kwargs

    @log_call(level='debug')
    def another_function(self, *args, **kwargs):
        return None

    ...
```

  You can just write:

```
from pydecor import log_call

@log_call(level='debug')
class FullyLoggedClass(object):

    def some_function(self, *args, **kwargs):
        return args, kwargs

    def another_function(self, *args, **kwargs):
        return None

    ...
```

  PyDecor ignores special methods (like `__init__`) so as not to interfere with deep Python magic. By default, it works on any methods of a class, including instance, class and static methods. It also ensures that class attributes are preserved after decoration, so your class references continue to behave as expected.

- **Consistent Method Decoration!**

  Whether you're decorating a class, an instance method, a class method, or a static method, you can use the same passed function. `self` and `cls` variables are stripped out of the method parameters passed to the provided callable, so your functions don't need to care about where they're used.

- **Lots of Tests!**

  Seriously. Don't believe me? Just look. We've got the best tests. Just phenomenal.

2.4 Installation

'pydecor' 2.0 and forward supports only Python 3.6+!

If you need support for an older Python, use the most recent 1.x release.

To install *pydecor*, simply run:

```
pip install -U pydecor
```

To install the current development release:

```
pip install --pre -U pydecor
```

You can also install from source to get the absolute most recent code, which may or may not be functional:

```
git clone https://github.com/mplanchard/pydecor
pip install ./pydecor
```

2.5 Details

2.5.1 Provided Decorators

This package provides generic decorators, which can be used with any function to provide extra utility to decorated resources, as well as prête-à-porter (ready-to-wear) decorators for immediate use.

While the information below is enough to get you started, I highly recommend checking out the decorator module docs to see all the options and details for the various decorators!

Generics

- `@before` - run a callable before the decorated function executes
  - called with an instance of *Decorated* and any provided kwargs
- `@after` - run a callable after the decorated function executes
  - called with an instance of *Decorated* and any provided kwargs
- `@instead` - run a callable in place of the decorated function
  - called with an instance of *Decorated* and any provided kwargs
- `@decorate` - specify multiple callables to be run before, after, and/or instead of the decorated function
  - callables passed to `decorate`'s `before`, `after`, or `instead` keyword arguments will be called with the same default function signature as described for the individual decorators, above. Extras will be passed to all provided callables.
- `construct_decorator` - specify functions to be run `before`, `after`, or `instead` of decorated functions. Returns a reusable decorator.

The callable passed to a generic decorator is expected to handle at least one positional argument, which will be an instance of *Decorated*. *Decorated* objects provide the following interface:

Attributes:

- *args*: a tuple of any positional arguments with which the decorated callable was called
- *kwargs*: a dict of any keyword arguments with which the decorated callable was called
- *wrapped*: a reference to the decorated callable
- *result*: when the _wrapped_ function has been called, its return value is stored here

Methods

- *__call__(*args, **kwargs)*: a shortcut to *decorated.wrapped(*args, **kwargs)*, calling an instance of *Decorated* calls the underlying wrapped callable. The result of this call (or a direct call to *decorated.wrapped()*) will set the *result* attribute.

Every generic decorator may take any number of keyword arguments, which will be passed directly into the provided callable, so, running the code below prints "red":

```
from pydecor import before

def before_func(_decorated, label=None):
    print(label)

@before(before_func, label='red')
def red_function():
    pass

red_function()
```

Every generic decorator takes the following keyword arguments:

- `implicit_method_decoration` - if True, decorating a class implies decorating all of its methods. **Caution:** you should probably leave this on unless you know what you are doing.
- `instance_methods_only` - if True, only instance methods (not class or static methods) will be automatically decorated when `implicit_method_decoration` is True

The `construct_decorator` function can be used to combine `@before`, `@after`, and `@instead` calls into one decorator, without having to worry about unintended stacking effects. Let's make a decorator that announces when we're starting an exiting a function:

```
from pydecor import construct_decorator

def before_func(decorated):
    print('Starting decorated function '
          '"{}"'.format(decorated.wrapped.__name__))

def after_func(decorated):
    print('"{}" gave result "{}"'.format(
        decorated.wrapped.__name__, decorated.result
    ))

my_decorator = construct_decorator(before=before_func, after=after_func)

@my_decorator()
def this_function_returns_nothing():
    return 'nothing'
```

And the output?

```
Starting decorated function "this_function_returns_nothing"
"this_function_returns_nothing" gave result "nothing"
```

Maybe a more realistic example would be useful. Let's say we want to add headers to a Flask response.

```
from flask import Flask, Response, make_response
from pydecor import construct_decorator
```

(continues on next page)

(continued from previous page)

```
def _set_app_json_header(decorated):
    # Ensure the response is a Response object, even if a tuple was
    # returned by the view function.
    response = make_response(decorated.result)
    response.headers.set('Content-Type', 'application/json')
    return response


application_json = construct_decorator(after=_set_app_json_header)


# Now you can decorate any Flask view, and your headers will be set.

app = Flask(__name__)

# Note that you must decorate "before" (closer to) the function than the
# app.route() decoration, because the route decorator must be called on
# the "finalized" version of your function

@app.route('/')
@application_json()
def root_view():
    return 'Hello, world!'

client = app.test_client()
response = app.get('/')

print(response.headers)
```

The output?

..code:

```
Content-Type: application/json
Content-Length: 13
```

Prête-à-porter (ready-to-wear)

- `export` - add the decorated class or function to its module's *__all__* list, exposing it as a "public" reference.
- `intercept` - catch the specified exception and optionally re-raise and/or call a provided callback to handle the exception
- `log_call` - automatically log the decorated function's call signature and results
- `memoize` - memoize a function's call and return values for re-use. Can use any cache in `pydecor.caches`, which all have options for automatic pruning to keep the memoization cache from growing too large.

2.5.2 Caches

Three caches are provided with `pydecor`. These are designed to be passed to the `@memoization` decorator if you want to use something other than the default `LRUCache`, but they are perfectly functional for use elsewhere.

All caches implement the standard dictionary interface.

LRUCache

A least-recently-used cache. Both getting and setting of key/value pairs results in their having been considered most-recently-used. When the cache reaches the specified `max_size`, least-recently-used items are discarded.

FIFOCache

A first-in, first-out cache. When the cache reaches the specified `max_size`, the first item that was inserted is discarded, then the second, and so on.

TimedCache

A cache whose entries expire. If a `max_age` is specified, any entries older than the `max_age` (in seconds) will be considered invalid, and will be removed upon access.

2.5.3 Stacking

Generic and ready-to-wear decorators may be stacked! You can stack multiple of the same decorator, or you can mix and match. Some gotchas are listed below.

Generally, stacking works just as you might expect, but some care must be taken when using the `@instead` decorator, or `@intercept`, which uses `@instead` under the hood.

Just remember that `@instead` replaces everything that comes before. So, as long as `@instead` calls the decorated function, it's okay to stack it. In these cases, it will be called *before* any decorators specified below it, and those decorators will be executed when it calls the decorated function. `@intercept` behaves this way, too.

If an `@instead` decorator does *not* call the decorated function and instead replaces it entirely, it **must** be specified first (at the bottom of the stacked decorator pile), otherwise the decorators below it will not execute.

For `@before` and `@after`, it doesn't matter in what order the decorators are specified. `@before` is always called first, and `@after` last.

2.5.4 Class Decoration

Class decoration is difficult, but PyDecor aims to make it as easy and intuitive as possible!

By default, decorating a class applies that decorator to all of that class' methods (instance, class, and static). The decoration applies to class and static methods whether they are referenced via an instance or via a class reference. "Extras" specified at the class level persist across calls to different methods, allowing for things like a class level memoization dictionary (there's a very basic test in the test suite that demonstrates this pattern).

If you'd prefer that the decorator not apply to class and static methods, set the `instance_methods_only=True` when decorating the class.

If you want to decorate the class itself, and *not* its methods, keep in mind that the decorator will be triggered when the class is instantiated, and that, if the decorator replaces or alters the return, that return will replace the instantiated class. With those caveats in mind, setting `implicit_method_decoration=False` when decorating a class enables that functionality.

Note: Class decoration, and in particular the decoration of class and static methods, is accomplished through some pretty deep, complicated magic. The test suite has a lot of tests trying to make sure that everything works as expected, but please report any bugs you find so that I can resolve them!

2.5.5 Method Decoration

Decorators can be applied to static, class, or instance methods directly, as well. If combined with `@staticmethod` or `@classmethod` decorators, those decorators should always be at the "top" of the decorator stack (furthest from the function).

When decorating instance methods, `self` is removed from the parameters passed to the provided callable.

When decorating class methods, `cls` is removed from the parameters passed to the provided callable.

Currently, the class and instance references *do not* have to be named `"cls"` and `"self"`, respectively, in order to be removed. However, this is not guaranteed for future releases, so try to keep your naming standard if you can (just FYI, `"self"` is the more likely of the two to wind up being required).

2.6 Examples

Below are some examples for the generic and standard decorators. Please check out the API Docs for more information, and also check out the convenience decorators, which are all implemented using the `before`, `after`, and `instead` decorators from this library.

2.6.1 Update a Function's Args or Kwargs

Functions passed to `@before` can either return None, in which case nothing happens to the decorated functions parameters, or they can return a tuple of args (as a tuple) and kwargs (as a dict), in which case those parameters are used in the decorated function. In this example, we sillify a very serious function.

Note: Because kwargs are mutable, they can be updated even if the function passed to *before* doesn't return anything.

```
from pydecor import before

def spamify_func(decorated):
    """Mess with the function arguments"""
    args = tuple(['spam' for _ in decorated.args])
    kwargs = {k: 'spam' for k in decorated.kwargs}
    return args, kwargs


@before(spamify_func)
def serious_function(serious_string, serious_kwarg='serious'):
    """A very serious function"""
    print('A serious arg: {}'.format(serious_string))
    print('A serious kwarg: {}'.format(serious_kwarg))

serious_function('Politics', serious_kwarg='Religion')
```

The output?

```
A serious arg: spam
A serious kwarg: spam
```

2.6.2 Do Something with a Function's Return Value

Functions passed to `@after` receive the decorated function's return value as part of the *Decorated* instance. If `@after` returns None, the return value is sent back unchanged. However, if `@after` returns something, its return value is sent back as the return value of the function.

In this example, we ensure that a function's return value has been thoroughly spammified.

```
from pydecor import after

def spamify_return(decorated):
    """Spamify the result of a function"""
    return 'spam-spam-spam-spam-{}-spam-spam-spam-spam'.format(decorated.result)


@after(spamify_return)
def unspammed_function():
    """Return a non-spammy value"""
    return 'beef'

print(unspammed_function())
```

The output?

```
spam-spam-spam-spam-beef-spam-spam-spam-spam
```

2.6.3 Do Something Instead of a Function

Functions passed to `@instead` also provide wrapped context via the *Decorated* object. But, if the *instead* callable does not call the wrapped function, it won't get called at all. Maybe you want to skip a function when a certain condition is True, but you don't want to use `pytest.skipif`, because `pytest` can't be a dependency of your production code for whatever reason.

```
from pydecor import instead

def skip(decorated, when=False):
    if when:
        pass
    else:
        # Calling `decorated` calls the wrapped function.
        return decorated(*decorated.args, **decorated.kwargs)


@instead(skip, when=True)
def uncalled_function():
    print("You won't see me (you won't see me)")


uncalled_function()
```

The output?

(There is no output, because the function was skipped)

2.6.4 Automatically Log Function Calls and Results

Maybe you want to make sure your functions get logged without having to bother with the logging boilerplate each time. `@log_call` tries to automatically get a logging instance corresponding to the module in which the decoration occurs (in the same way as if you made a call to `logging.getLogger(__name__)`, or you can pass it your own, fancy, custom, spoiler-bedecked logger instance.

```
from logging import getLogger, StreamHandler
from sys import stdout

from pydecor import log_call


# We're just getting a logger here so we can see the output. This isn't
# actually necessary for @log_call to work!
log = getLogger(__name__)
log.setLevel('DEBUG')
log.addHandler(StreamHandler(stdout))


@log_call()
def get_lucky(*args, **kwargs):
    """We're up all night 'till the sun."""
    return "We're up all night for good fun."


get_lucky('Too far', 'to give up', who_we='are')
```

And the output?

```
get_lucky(*('Too far', 'to give up'), **{'who_we': 'are'}) -> "We're up all night for␣
↪good fun"
```

2.6.5 Intercept an Exception and Re-raise a Custom One

Are you a put-upon library developer tired of constantly having to re-raise custom exceptions so that users of your library can have one nice try/except looking for your base exception? Let's make that easier:

```
from pydecor import intercept


class BetterException(Exception):
    """Much better than all those other exceptions"""


@intercept(catch=RuntimeError, reraise=BetterException)
def sometimes_i_error(val):
    """Sometimes, this function raises an exception"""
    if val > 5:
        raise RuntimeError('This value is too big!')


for i in range(7):
    sometimes_i_error(i)
```

The output?

```
Traceback (most recent call last):
  File "/Users/Nautilus/Library/Preferences/PyCharm2017.1/scratches/scratch_1.py",␣
↪line 88, in <module>
    sometimes_i_error(i)
  File "/Users/Nautilus/Documents/Programming/pydecor/pydecor/decorators.py", line␣
↪389, in wrapper
    return fn(**fkwargs)
  File "/Users/Nautilus/Documents/Programming/pydecor/pydecor/functions.py", line 58,␣
↪in intercept
    raise_from(new_exc, context)
  File "<string>", line 2, in raise_from
__main__.BetterException: This value is too big!
```

2.6.6 Intercept an Exception, Do Something, and Re-raise the Original

Maybe you don't *want* to raise a custom exception. Maybe the original one was just fine. All you want to do is print a special message before re-raising the original exception. PyDecor has you covered:

```
from pydecor import intercept


def print_exception(exc):
    """Make sure stdout knows about our exceptions"""
    print('Houston, we have a problem: {}'.format(exc))


@intercept(catch=Exception, handler=print_exception, reraise=True)
def assert_false():
    """All I do is assert that False is True"""
    assert False, 'Turns out, False is not True'


assert_false()
```

And the output:

```
Houston, we have a problem: Turns out, False is not True
Traceback (most recent call last):
  File "/Users/Nautilus/Library/Preferences/PyCharm2017.1/scratches/scratch_1.py",␣
↪line 105, in <module>
    assert_false()
  File "/Users/Nautilus/Documents/Programming/pydecor/pydecor/decorators.py", line␣
↪389, in wrapper
    return fn(**fkwargs)
  File "/Users/Nautilus/Documents/Programming/pydecor/pydecor/functions.py", line 49,␣
↪in intercept
    return decorated(*decorated_args, **decorated_kwargs)
  File "/Users/Nautilus/Library/Preferences/PyCharm2017.1/scratches/scratch_1.py",␣
↪line 102, in assert_false
    assert False, 'Turns out, False is not True'
AssertionError: Turns out, False is not True
```

2.6.7 Intercept an Exception, Handle, and Be Done with It

Sometimes an exception isn't the end of the world, and it doesn't need to bubble up to the top of your application. In these cases, maybe just handle it and don't re-raise:

```
from pydecor import intercept


def let_us_know_it_happened(exc):
    """Just let us know an exception happened (if we are reading stdout)"""
    print('This non-critical exception happened: {}'.format(exc))


@intercept(catch=ValueError, handler=let_us_know_it_happened)
def resilient_function(val):
    """I am so resilient!"""
    val = int(val)
    print('If I get here, I have an integer: {}'.format(val))


resilient_function('50')
resilient_function('foo')
```

Output:

```
If I get here, I have an integer: 50
This non-critical exception happened: invalid literal for int() with base 10: 'foo'
```

Note that the function does *not* continue running after the exception is handled. Use this for short-circuiting under certain conditions rather than for instituting a `try/except:pass` block. Maybe one day I'll figure out how to make this work like that, but as it stands, the decorator surrounds the entire function, so it does not provide that fine-grained level of control.

2.7 Roadmap

2.7.1 2.?

More Prête-à-porter Decorators

- `skipif` - similar to py.test's decorator, skip the function if a provided condition is True

Let me know if you've got any idea for other decorators that would be nice to have!

Type Annotations

Now that we've dropped support for Python 2, we can use type annotations to properly annotate function inputs and return values and make them available to library authors.

2.8 Contributing

Contributions are welcome! If you find a bug or if something doesn't work the way you think it should, please raise an issue. If you know how to fix the bug, please open a PR!

I absolutely welcome any level of contribution. If you think the docs could be better, or if you've found a typo, please open up a PR to improve and/or fix them.

2.8.1 Contributor Conduct

There is a `CODE_OF_CONDUCT.md` file with details, based on one of GitHub's templates, but the upshot is that I expect everyone who contributes to this project to do their best to be helpful, friendly, and patient. Discrimination of any kind will not be tolerated and will be promptly reported to GitHub.

On a personal note, Open Source survives because of people who are willing to contribute their time and effort for free. The least we can do is treat them with respect.

2.8.2 Tests

Tests can be run with:

```
make test
```

This will use whatever your local *python3* happens to be. If you have other pythons available, you can run:

```
make tox
```

to try to run locally for all supported Python versions.

If you have docker installed, you can run:

```
make test-docker-{version}  # e.g. make test-docker-3.6
```

to pull down an appropriate Docker image and run tests inside of it. You can also run:

```
make test-docker
```

to do this for all supported versions of Python.

PRs that cause tests to fail will not be merged until tests pass.

Any new functionality is expected to come with appropriate tests. If you have any questions, feel free to reach out to me via email at `msplanchard @ gmail` or on GH via Issues.

2.8.3 Autoformatting

This project uses black for autoformatting. I recommend setting your editor up to format on save for this project, but you can also run:

```
make fmt
```

to format everything.

2.8.4 Linting

Linting can be run with:

```
make lint
```

Currently, linting verifies that there are:

- No flake8 errors
- No mypy errors
- No pylint errors
- No files that need be formatted

You should ensure that *make lint* returns 0 before opening a PR.

2.8.5 Docs

Docs are autogenerated via Sphinx. You can build them locally by running:

```
make docs
```

You can then open *docs/_build/html/index.html* in your web browser of choice to see how the documentation will look with your changes.

2.8.6 Deployment

Deployment is handled through pushing tags. Any tag pushed to GH causes a push to PyPI if the current version is not yet present there.

Pushing the appropriate tag is made easier through the use of:

```
VERSION=1.0.0 make distribute
```

where *VERSION* obviously should be the current version. This will verify the specified version matches the package's current version, check to be sure that the most recent master is being distributed, prompt for a message, and create and push a signed tag of the format *v{version}*.

2.9 Credits and Links

- This project was started using my generic project template
- Tests are run with pytest and tox
- Documentation built with sphinx
- Coverage information collected with coverage
- Pickling of objects provided via dill

CHAPTER 3

CHANGELOG

CHAPTER 4

Changelog

4.1 2.0.0

- `export` - register entities in a module's *__all__* list (thanks @Paebbels!)
- Use of `Decorator` class and consistent callable signature for generic decorators is now required
- Drop support for Python <3.6
- Move to a *src/* layout
- Lots of clarifications, typo fixes, and improvements to the docs
- Lots of development environment improvements: * Automatic distribution of tagged commits via PyPI (thanks @Paebbels!) * `Makefile` for a consistent interface into build operations * Improvements to `tox` configuration * Addition of consistent and required linting with pylint, mypy, and flake8 * Autoformatting with `black`

4.2 1.1.3

Apparently *pythonhosted.org* has been deprecated, so I set up a Read the Docs account and moved the documentation there.

- Uploaded README to point to new docs
- Added docs build image for funsies.

4.3 1.1.2

- fixed an issue with the README

4.4 1.1.1

- fixed setup to pull README once more, even in Python 2

4.5 1.1.0

Memoization, prep for v 2.0

- `memoize` decorator - memoize any callable
- `LRUCache`, `FIFOCache`, and `TimedCace` - to make the `memoize` decorator more useful
- `Decorated` class, prep for v 2.0
- `_use_future_syntax` option, prepping for version 2.0

4.6 1.0.0

Initial release!

- `before` decorator - run a callback prior to the decorated function
- `after` decorator - run a callback after the decorated function
- `instead` decorator - run a callback instead of the decorated function
- `decorate` decorator - specify before, after, and/or instead callbacks all at once
- `construct_decorator` function - create a reusable decorator with before, after, and/or instead callbacks
- `intercept` decorator - wrap the decorated function in a try/except, specifying a function with which to handle the exception and/or another exception to re-raise
- `log_call` decorator - automatically log the decorated functions's call signature and results

CHAPTER 5

Indices and tables

- genindex
- modindex
- search