metawrap Documentation

Release 0.0.1+12.ga37e365.dirty

John Kirkham

Mar 09, 2017

Contents

Contents:

CHAPTER 1

metawrap

A collection of wrappers for functions and classes.

- Free software: BSD 3-Clause
- Documentation: https://metawrap.readthedocs.io.

Features

- TODO

Credits

This package was created with Cookiecutter and the nanshe-org/nanshe-cookiecutter project template.

CHAPTER 2

Installation

Stable release

To install metawrap, run this command in your terminal:

```
$ pip install metawrap
```

This is the preferred method to install metawrap, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

From sources

The sources for metawrap can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/jakirkham/metawrap
```

Or download the tarball:

```
$ curl  -OL https://github.com/jakirkham/metawrap/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

Usage

To use metawrap in a project:

```
import metawrap
```

CHAPTER 4

API

metawrap package

Submodules

metawrap.metawrap module

The module `metawrap` provides support decorating functions and classes.

Overview

The module `metawrap` extends wrapping abilities found in `functools`. In particular, it is ensured all wrapped functions contain an attribute `__wrapped__`, which points back to the original function before the wrapper was applied. Also, the ability to wrap classes with a decorator to apply a `metaclass` or series of ‘‘metaclass‘‘es is provided. Making it much easier to transform classes without mucking in their internals.

API

`metawrap.metawrap.`**`class_decorate_all_methods`**(**decorators*)

Returns a decorator that decorates a class such that all its methods are decorated by the decorators provided.

Parameters **`*decorators`** (*`tuple`*) – decorators to decorate all methods with.

Returns a decorator for the class.

Return type (decorator)

`metawrap.metawrap.`**`class_decorate_methods`**(***method_decorators*)

Returns a decorator that decorates a class such that specified methods are decorated by the decorators provided.

Parameters **`**method_decorators`** (*`tuple`*) – method names with a single decorator or a list of decorators.

Returns a decorator for the class.

Return type (decorator)

metawrap.metawrap.**class_static_variables**(***kwargs*)
Returns a decorator that decorates a class such that it has the given static variables set.

Parameters ****kwargs** (*tuple*) – keyword args will be set to the value provided.

Returns a decorator for the class.

Return type (decorator)

metawrap.metawrap.**identity_wrapper**(*a_callable*)
Trivially wraps a given callable without doing anything else to it.

Parameters **a_callable** (*callable*) – the callable that is being wrapped.

Returns a wrapped callable.

Return type (callable)

metawrap.metawrap.**metaclass**(*meta*)
Returns a decorator that decorates a class such that the given metaclass is applied.

Note: Decorator will add the __metaclass__ attribute so the last metaclass applied is known. Also, decorator will add the __wrapped__ attribute so that the unwrapped class can be retrieved.

Parameters **meta** (metaclass) – metaclass to apply to a given class.

Returns a decorator for the class.

Return type (decorator)

metawrap.metawrap.**metaclasses**(**metas*)
Returns a decorator that decorates a class such that the given metaclasses are applied.

Note: Shorthand for repeated application of metaclass.

Parameters ***metas** (metaclasses) – metaclasses to apply to a given class.

Returns a decorator for the class.

Return type (decorator)

metawrap.metawrap.**repack_call_args**(*a_callable*, **args*, ***kwargs*)
Reorganizes args and kwargs to match the given callables signature.

Parameters

- **a_callable** (*callable*) – some callable.
- ***args** (*callable*) – positional arguments for the callable.
- ****kwargs** (*callable*) – keyword arguments for the callable.

Returns

all arguments as passed as position arguments, all default arguments and all arguments passed as keyword arguments.

Return type args (tuple)

metawrap.metawrap.**static_variables**(***kwargs*)
Returns a decorator that decorates a callable such that it has the given static variables set.

Parameters ***kwargs** (*tuple*) – keyword args will be set to the value provided.

Returns a decorator for the callable.

Return type (decorator)

metawrap.metawrap.**tied_call_args**(*a_callable*, **args*, ***kwargs*)
Ties all the args to their respective variable names.

Parameters

- **a_callable** (*callable*) – some callable.
- ***args** (*callable*) – positional arguments for the callable.
- ****kwargs** (*callable*) – keyword arguments for the callable.

Returns

ordered dictionary of arguments name and their values, all variadic position arguments, all variadic keyword arguments.

Return type args (tuple)

metawrap.metawrap.**unwrap**(*a_callable*)
Returns the underlying function that was wrapped.

Parameters **a_callable** (*callable*) – some wrapped (or not) callable.

Returns the callable that is no longer wrapped.

Return type (callable)

metawrap.metawrap.**update_wrapper**(*wrapper*, *wrapped*, *assigned=('__module__', '__name__', '__doc__')*, *updated=('__dict__',)*)
Extends functools.update_wrapper to ensure that it stores the wrapped function in the attribute __wrapped__.

Parameters

- **wrapper** (*callable*) – the replacement callable.
- **wrapped** (*callable*) – the callable that is being wrapped.
- **assigned** (*tuple*) – is a tuple naming the attributes assigned directly from the wrapped function to the wrapper function (defaults to functools.WRAPPER_ASSIGNMENTS)
- **updated** (*tuple*) – is a tuple naming the attributes of the wrapper that are updated with the corresponding attribute from the wrapped function (defaults to functools.WRAPPER_UPDATES)

Returns the wrapped callable.

Return type (callable)

metawrap.metawrap.**with_setup_state**(*setup=None*, *teardown=None*)
Adds setup and teardown callable to a function s.t. they can mutate it.

Based on `with_setup` from `nose`. This goes a bit further than `nose` does and provides a mechanism for the setup and teardown functions to change the callable in question. In other words, variables generated in setup can be stored in the functions globals and then cleaned up and removed in teardown. The final result of using this function should be a function equivalent to one generated by `with_setup`.

Parameters

- **setup** (*callable*) – A callable that takes the decorated function as an argument. This sets up the function before execution.
- **teardown** (*callable*) – A callable that takes the decorated function as an argument. This cleans up the function after execution.

Returns Does the actual decoration.

Return type callable

metawrap.metawrap.**with_setup_state_handler**(*a_callable*)

A final wrapper for `with_setup_state`.

This calls `setup` and `teardown` before and after if defined. When used as a decorator, this should come after all `setup` and `teardown` calls.

Parameters **a_callable** (*callable*) – A callable to run `setup` and `teardown` on.

Returns The wrapped function.

Return type callable

metawrap.metawrap.**wraps**(*wrapped*, *assigned=('__module__', '__name__', '__doc__')*, *updated=('__dict__',)*)

Builds on functools.wraps to ensure that it stores the wrapped function in the attribute __wrapped__.

Parameters

- **wrapped** (*callable*) – the callable that is being wrapped.
- **assigned** (*tuple*) – is a tuple naming the attributes assigned directly from the wrapped function to the wrapper function (defaults to functools.WRAPPER_ASSIGNMENTS)
- **updated** (*tuple*) – is a tuple naming the attributes of the wrapper that are updated with the corresponding attribute from the wrapped function (defaults to functools.WRAPPER_UPDATES)

Returns

a decorator for callable, which will contain wrapped.

Return type (callable)

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

Types of Contributions

Report Bugs

Report bugs at https://github.com/jakirkham/metawrap/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

Write Documentation

metawrap could always use more documentation, whether as part of the official metawrap docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/jakirkham/metawrap/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

Get Started!

Ready to contribute? Here's how to set up *metawrap* for local development.

1. Fork the *metawrap* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/metawrap.git
```

3. Install your local copy into an environment. Assuming you have conda installed, this is how you set up your fork for local development (on Windows drop *source*). Replace *"<some version>"* with the Python version used for testing.:

```
$ conda create -n metawrapenv python="<some version>"
$ source activate metawrapenv
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions:

```
$ flake8 metawrap tests
$ python setup.py test or py.test
```

   To get flake8, just conda install it into your environment.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5, and 3.6. Check https://travis-ci.org/jakirkham/metawrap/pull_requests and make sure that the tests pass for all supported Python versions.

Tips

To run a subset of tests:

```
$ python -m unittest tests.test_metawrap
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index