singletons

Dec 06, 2019

Contents

CHAPTER 1

Overview

docs	
tests	
package	

Declaring singleton classes and singleton factories with different scopes of instantiation, striving for thread-safety and simplicity.

- Free software: MIT license

1.1 Installation

```
pip install singletons
```

1.2 Quick Example

```
import singletons

@singletons.GlobalFactory
def my_uuid():
    return uuid.uuid4()

# elsewhere...
my_uuid()  # will return the global instance of a UUID object
```

1.3 Documentation

https://python-singletons.readthedocs.io/

1.4 Development

To run the all tests run:

```
tox
```

CHAPTER 2

Usage

One of the simplest ways to use `singletons` is using a factory decorator to make the return value of a function a singleton object. Create a `shared.py` file:

```
import uuid
import singletons

@singletons.GlobalFactory
def my_uuid():
    return uuid.uuid4()
```

Any time you want to access the instance generated by your factory, just call the `my_uuid()` function.

Factory decorators include:

- `GlobalFactory`
- `ProcessFactory`
- `ThreadFactory`
- `GreenthreadFactory`
- `EventletFactory`
- `GeventFactory`

You can also declare a class as a singleton by using the `metaclass` keyword argument:

```
import singletons

class SharedCache(dict, metaclass=singletons.ThreadSingleton):
    pass
```

When `SharedCache` is called (using `SharedCache()`), if an object already exists for the current thread it is returned, otherwise it is constructed.

Singleton metaclasses include:

- `Singleton`
- `ProcessSingleton`
- `ThreadSingleton`
- `GreenthreadSingleton`
- `EventletSingleton`
- `GeventSingleton`

2.1 Writing Tests

A common need when working with singletons is to be able to use Mock objects for unit tests. `singletons` includes a helper class for making modules easily swappable to use Mocks for everything instead of the factories/classes defined. A common usage would be to put these lines at the bottom of your `shared.py` file:

```
class _Shared(singletons.SharedModule):
    globals = globals()
sys.modules[__name__] = _Shared()
```

To enable the Mock object replacement, call `setup_mock()` or set the environment variable `SINGLETONS_SETUP_MOCK=1`. This will replace all accesses of module attributes with Mock() instances. `setup_mock` can be called inside a TestCase `setup()` method or as part of a pytest fixture to ensure that each test has a clean set of Mock() instances.

Example test:

```
class MyTestCase(unittest.TestCase):
    def setup(self):
        shared.setup_mock()

    def test_get_documents():
        c = shared.session()
        # do thing
        c.request.assert_called_once()
```

To use custom Mock objects, set them as attributes on the module after calling `setup_mock`:

```
class MyTestCase(unittest.TestCase):
    def setup(self):
        shared.setup_mock()
        mock_instance = mock.Mock(spec=User)
        mock_instance.name = 'Jane Doe'
        mock_instance.username = 'jdoe123'
        shared.mock_instance = mock_instance

    def test_get_userdata():
        c = shared.mock_instance()
        # do thing
        c.request.assert_called_once()
```

CHAPTER 3

Reference

3.1 singletons

CHAPTER 4

Contributing

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

4.1 Bug reports

When 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.

4.2 Documentation improvements

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

4.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/jmaroeder/python-singletons/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 code contributions are welcome :)

4.4 Development

To set up `python-singletons` for local development:

1. Fork python-singletons (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/python-singletons.git
```

3. Create a branch for local development:

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

   Now you can make your changes locally.

4. When you're done making changes, run all the checks, doc builder and spell checker with tox one command:

```
tox
```

5. 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
```

6. Submit a pull request through the GitHub website.

4.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

4.4.2 Tips

To run a subset of tests:

```
tox -e envname -- py.test -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though ...

CHAPTER 5

Authors

- James Roeder - https://www.jroeder.net

CHAPTER 6

Changelog

6.1 master

6.2 0.2.2 (2018-02-01)

- Shows warning rather than giving exception when using greenthread singletons without a greenthread environment

6.3 0.2.1 (2018-01-29)

- CI changes

6.4 0.2.0 (2018-01-23)

- More tests
- Usage examples added to documentation
- Fixed issues revealed by tests

6.5 0.1.0 (2018-01-22)

- First release on PyPI.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search