marrydoc Documentation

Release 0.1.0

Daniel Mark Gass

May 24, 2018

Contents

The `marrydoc` Python module makes maintaining consistency of related Python docstrings easy. `marrydoc` provides decorators to "wed" a docstring to another and provides a command line tool to automatically update a module's docstrings when their basis docstrings have changed.

The decorators offered annotate class, function, and method docstrings to identify if their docstring is to be inherited from, maintained as a copy of, or maintained as a modified copy of a docstring of another program construct.

CHAPTER 1

Quick Start

1.1 @inherit

Use the `inherit()` decorator to dynamically copy a docstring from one program construct to another when a module is imported. For example:

```
import marrydoc
from foo import bar

@marrydoc.inherit(bar)
def my_bar():
    pass

assert bar.__doc__ == my_bar.__doc__
```

1.2 @copied_from

Use the `copied_from()` decorator in combination with the command line tool to evaluate if one program construct docstring is up to date with another and automatically update the script if they are unequal. For example:

```
import marrydoc
from foo import bar

@marrydoc.copied_from(bar)
def my_bar():
    """Perform foo bar."""
    pass
```

Then use the command line tool to evaluate if the source docstring has changed and automatically update if so:

```
$ python -m marrydoc --merge my_foo.py
my_foo.py ... OK
```

1.3 @based_on

Use the `based_on()` decorator instead of `copied_from()` when the docstring is a copy but has been modified. Pass an unmodified copy of the source docstring as the second argument to `based_on()` (to facilitate source docstring change detection and provide a basis of a three way merge). For example:

```
import marrydoc
from foo import bar

@marrydoc.based_on(
    bar,
    """Perform foo bar.""")
def my_bar():
    """Perform my special foo bar."""
    pass
```

Then use the command line tool to evaluate if the source docstring has changed and automatically perform a three way merge if so:

```
$ python -m marrydoc --merge my_foo.py
my_foo.py ... UPDATED
```

1.3.1 [baseline] About

Contributors

- **Dan Gass (dan.gass@gmail.com)**
  - Primary author

Development

Repository https://gitlab.com/dangass/marrydoc

License

```
MIT License

Copyright (c) 2018 Daniel Mark Gass (dan.gass@gmail.com)

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
```

(continues on next page)

(continued from previous page)

```
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

1.3.2 [MarryDoc] API Reference

Decorators

Command Line Tool

```
$ python -m marrydoc --help
usage: marrydoc [-h] [-m] [-v] [-w] path [path ...]

Ensure related docstrings are consistent.

positional arguments:
  path           module or directory path

optional arguments:
  -h, --help     show this help message and exit
  -m, --merge    update module docstrings
  -w, --walk     recursively walk directories
```

Note: After installation, a `marrydoc` "shim" executable exists in the `Scripts` subdirectory of your Python installation. If your operating system has been configured to include the `Scripts` subdirectory in the path, the tool may be invoked directly:

```
$ marrydoc --help
```

1.3.3 [MarryDoc] Installation

Prerequisites

- Python
  - version 2.7
  - version 3.4 or higher
- Requirements for Installing Packages (located in the Installing Packages tutorial within the Python Packaging User Guide).

Install Steps

At a shell prompt, use pip to automatically download and install *marrydoc*:

```
python -m pip install --upgrade marrydoc
```

1.3.4 [MarryDoc] Release Notes

Versions are incremented according to semver.

0.2

- **0.2.0 (2018-05-23)**
  - Support specifying module name on command line.

0.1

- **0.1.0 (2018-05-02)**
  - Initial "beta" release.

1.3.5 [MarryDoc] Usage

Inherit DocString

The `inherit()` decorator copies a docstring from a specified object and attaches it to the function or method it decorates. In the following example, standard library `b2a_hex()` function serves as the docstring source:

`inherit_example.py`.

```
import binascii
import marrydoc

@marrydoc.inherit(binascii.b2a_hex)
def b2a_hex(data, sep=''):
    mashed = binascii.b2a_hex(data)
    return sep.join(mashed[i:i+2] for i in range(0, len(mashed), 2))

hexlify = b2a_hex
```

The `inherit()` decorator guarentees that the docstrings are always synchronized and does not require maintenance of the module when the original docstring changes:

```
>>> import binascii
>>> import inherit_example
>>>
>>> print binascii.hexlify.__doc__
b2a_hex(data) -> s; Hexadecimal representation of binary data.

This function is also available as "hexlify()".
>>>
>>> print inherit_example.hexlify.__doc__
b2a_hex(data) -> s; Hexadecimal representation of binary data.

This function is also available as "hexlify()".
>>>
>>> binascii.hexlify.__doc__ == inherit_example.hexlify.__doc__
True
```

Maintain DocString Copies

The shortcoming of the `inherit()` decorator is that the copied docstring does not appear in the module and makes module maintenance more difficult. The `copied_from()` decorator defines a one to one relationship between the source of the docstring and the docstring of the function it decorates:

`copied_from_example.py`.

```
import binascii
import marrydoc


@marrydoc.copied_from(binascii.b2a_hex)
def b2a_hex(data, sep=''):
    """b2a_hex(data) -> s; Hexadecimal representation of binary data.

    This function is also available as "hexlify()"."""
    mashed = binascii.b2a_hex(data)
    return sep.join(mashed[i:i+2] for i in range(0, len(mashed), 2))

hexlify = b2a_hex
```

The defined relationship allows the *Command Line Tool* to check the module's docstring:

```
$ python -m marrydoc copied_from_example.py
copied_from_example.py ... OK
```

If the docstring source changed, the command line tool updates the docstring in the module when specifying the `--merge` option:

```
$ python -m marrydoc --merge copied_from_example.py
copied_from_example.py ... UPDATED
```

Note: During normal import of a module, the `copied_from()` decorator acts as a passthrough and introduces very little overhead.

Maintain DocString With Modifications

The `based_on()` decorator defines a relationship between the docstring of the function it decorates and the basis from which it was derived. The same as `copied_from()` and `inherit()`, the first argument to `based_on()` is the program construct containing the docstring that is to be tracked. `based_on()` requires a second argument that is a copy of the source docstring (the source docstring is compared against the copy and if they are unequal, the two values in combination with the actual docstring facilitate a three way merge). For example:

based_on_example.py.

```
import binascii
import marrydoc


@marrydoc.based_on(
    binascii.b2a_hex,
    """b2a_hex(data) -> s; Hexadecimal representation of binary data.

    This function is also available as "hexlify()".""")
def b2a_hex(data, sep=''):
    """b2a_hex(data) -> s; Hexadecimal representation of binary data.
    b2a_hex(data, sep) -> s; Separated hexadecimal representation of binary␣
↪data.

    This function is also available as "hexlify()"."""
    mashed = binascii.hexlify(data)
    return sep.join(mashed[i:i+2] for i in range(0, len(mashed), 2))

hexlify = b2a_hex
```

The defined relationship to the source in combination with the docstring copy provided as the second argument allow the *Command Line Tool* to check if the source docstring has changed:

```
$ python -m marrydoc based_on_example.py
based_on_example.py ... OK
```

If the source docstring has changed, the *Command Line Tool* updates the module's docstrings by performing a three way merge when specifying the `--merge` option:

```
$ python -m marrydoc --merge based_on_example.py
based_on_example.py ... UPDATED
```

Note: The three way merge requires a merge tool of your choosing. Without configuring, the three way merge attempts usage of `kdiff3`. See the next section for more information on configuring your favorite merge tool to be used.

During normal import of a module, the `based_on()` decorator acts as a passthrough and introduces very little overhead.

Configure Your Favorite Merge Tool

For three way merges, the `marrydoc` command line tool uses `kdiff3` when it is installed and in your system path. Otherwise `marrydoc` generates "base", "left", and "right" files on your file system for you to merge manually.

To automatically invoke your favorite three way merge tool instead, set the `MARRYDOC_MERGE` environment variable and specify the command line invocation using Python's string format substitution syntax.

For example, on Linux:

```
export MARRYDOC_MERGE="kdiff3 --merge --auto {base} {left} {right} --output
↪{orig}"
```

Or on Microsoft Windows:

```
set MARRYDOC_MERGE="kdiff3 --merge --auto {base} {left} {right} --output
↪{orig}"
```

Warning: The executable name/path must not contain spaces. The current implementation of `marrydoc` splits the string on whitespace and passes the result to a subprocess command.

Test DocStrings In A Module Collection

The `main()` function exposes the command line interface and offers a convenient method to check if the docstrings in a module are up to date. Add a test case within the module's regression test to call the command line interface and to check the returned exit code for success indication (0).

The `--walk` option is useful for checking an entire package hierarchy, for example:

```
import os
import unittest
import marrydoc
import mypackage

class TestDocStrings(unittest.TestCase):

    def test_package(self):
        package_path = os.path.dirname(mypackage.__file__)
        exitcode = marrydoc.main(['--walk', package_path])
        self.assertEqual(exitcode, 0)
```

Tips and Tricks

- `based_on()`, `copied_from()` may also be used to decorate a class to wed its docstring to another. (`inherit()` cannot be used to decorate a class because the class docstring is not settable by the time the docorator executes.)
- The `based_on()`, `copied_from()`, and `inherit()` decorators may also be used in combination with the `@classmethod()` or `staticmethod()` decorators. The `marrydoc` decorator implementations accomodate decorating in either order. For better readability, place the `@classmethod()` and `staticmethod()` decorators first (on the outside).
- When decorating a method using `based_on()`, `copied_from()`, or `inherit()`, a class may be passed as the first argument to specify the source of the docstring. The docstring of the method by the same name in the specified class then acts as the docstring basis.
- Ensure your operating system path includes the `Scripts` subdirectory that is part of the normal Python installation. After installation, a `marrydoc` "shim" executable exists that subdirectory to invoke the command line tool directly:

```
$ marrydoc --help
```

- The `marrydoc` command line also accepts importable module and package names. Use this form when the module is in the Python system path. This form may be used in combination with the `--walk` option to check an entire package, for example:

```
$ marrydoc baseline --walk
/usr/local/lib/python3.5/dist-packages/baseline/__about__.py ... OK
/usr/local/lib/python3.5/dist-packages/baseline/__init__.py ... OK
/usr/local/lib/python3.5/dist-packages/baseline/__main__.py ... OK
/usr/local/lib/python3.5/dist-packages/baseline/_baseline.py ... OK
/usr/local/lib/python3.5/dist-packages/baseline/_script.py ... OK
```