generator Documentation

Release 0.1.2

Kevin Stone

Aug 04, 2018

Contents

Generator is a helper for generating test methods for nose while still using unittest.

- Free software: ISC license
- Documentation: https://generator.readthedocs.org.

CHAPTER 1

Installation

```
pip install test-generator
```

CHAPTER 2

Introduction

Have you ever written tests that loop through a list of inputs to validate the functionality?

Something like?

```
from mything import thingy

class MyTestCase(unittest.TestCase):
    def test_thingy(self):
        for input in [
            'a',
            'b',
            'cccc',
            'ddd'
            'eeeeee',
            'f',
            'g'
        ]:
            self.assertTrue(thingy(input))
```

But running in a loop limits all the functionality in `TestCase` like per- test setUp or tearDown. It also fails on the first input and you can't run a single test input, you have to run them all? (Doesn't work well when each test is more complicated than this toy case).

Instead, what if you wrote your test like:

```
from generator import generator, generate
from mything import thingy

@generator
class MyTestCase(unittest.TestCase):

    @generate('a', 'b', 'cccc', 'ddd', 'eeeeee', 'f', 'g')
    def test_thingy(self, input):
        self.assertTrue(thingy(input))
```

And when you run your tests, you see:

```
----------------------------------------------------------------------
Ran 7 tests in 0.001s

OK
```

Generator gives you simple decorators to mulitply your test methods based on an argument list. It's great for checking a range of inputs, a list of error conditions or expected status codes.

CHAPTER 3

Examples

3.1 API Client Error Handling

Let's make sure our API client properly handles error conditions and raises a generic APIError under the conditions. We'll use mock to patch out the actual API call to return our response.

```
import mock
from generator import generator, generate
from example import client, APIError

@generator
class TestAPIErrorHandling(unittest.TestCase):

    @generate(400, 401, 403, 404, 500, 502, 503)
    def test_error(self, status_code):
        with mock.patch(client, '_request') as _request_stub:
            _request_stub.return_value.status_code = status_code

            self.assertRaises(APIError):
                client.get('/path/')
```

3.2 Test Fixtures

Let's make sure our API client properly handles error conditions and raises a generic APIError under the conditions. We'll use mock to patch out the actual API call to return our response.

```
from generator import generator, generate
from example.sanitize import strip_tags

@generator
class TestStripTags(unittest.TestCase):
```

(continues on next page)

(continued from previous page)

```
    @generate(
        ('<h1>hi</h1>', 'hi'),
        ('<script></script>something', 'something'),
        ('<div class="important"><p>some text</p></div>', 'some text'),
    )
    def test_strip_tags(self, input, expected):
        self.assertEqual(strip_tags(input), expected)
```

CHAPTER 4

Contents, indices and tables

4.1 Generator

Generator is a helper for generating test methods for nose while still using unittest.

- Free software: ISC license
- Documentation: https://generator.readthedocs.org.

4.1.1 Installation

```
pip install test-generator
```

4.1.2 Introduction

Have you ever written tests that loop through a list of inputs to validate the functionality?

Something like?

```
from mything import thingy

class MyTestCase(unittest.TestCase):
    def test_thingy(self):
        for input in [
            'a',
            'b',
            'cccc',
            'ddd'
            'eeeeee',
            'f',
            'g'
```

(continues on next page)

(continued from previous page)

```
        ]:
            self.assertTrue(thingy(input))
```

But running in a loop limits all the functionality in `TestCase` like per- test setUp or tearDown. It also fails on the first input and you can't run a single test input, you have to run them all? (Doesn't work well when each test is more complicated than this toy case).

Instead, what if you wrote your test like:

```
from generator import generator, generate
from mything import thingy

@generator
class MyTestCase(unittest.TestCase):

    @generate('a', 'b', 'cccc', 'ddd', 'eeeeee', 'f', 'g')
    def test_thingy(self, input):
        self.assertTrue(thingy(input))
```

And when you run your tests, you see:

```
----------------------------------------------------------------------
Ran 7 tests in 0.001s

OK
```

Generator gives you simple decorators to mulitply your test methods based on an argument list. It's great for checking a range of inputs, a list of error conditions or expected status codes.

4.1.3 Examples

API Client Error Handling

Let's make sure our API client properly handles error conditions and raises a generic APIError under the conditions. We'll use mock to patch out the actual API call to return our response.

```
import mock
from generator import generator, generate
from example import client, APIError

@generator
class TestAPIErrorHandling(unittest.TestCase):

    @generate(400, 401, 403, 404, 500, 502, 503)
    def test_error(self, status_code):
        with mock.patch(client, '_request') as _request_stub:
            _request_stub.return_value.status_code = status_code

            self.assertRaises(APIError):
                client.get('/path/')
```

Test Fixtures

Let's make sure our API client properly handles error conditions and raises a generic APIError under the conditions. We'll use mock to patch out the actual API call to return our response.

```
from generator import generator, generate
from example.sanitize import strip_tags

@generator
class TestStripTags(unittest.TestCase):

    @generate(
        ('<h1>hi</h1>', 'hi'),
        ('<script></script>something', 'something'),
        ('<div class="important"><p>some text</p></div>', 'some text'),
    )
    def test_strip_tags(self, input, expected):
        self.assertEqual(strip_tags(input), expected)
```

4.2 Installation

At the command line:

```
pip install test-generator
```

4.3 Usage

You can use Generator as either a decorator or a mixin. The decorator is a bit cleaner, but doesn't automatically generate any decorated methods in a sub-class.

4.3.1 Decorator

```
import unittest
from generator import generate, generator


@generator
class MyTestCase(unittest.TestCase):

    @generate(1, 2, 3):
    def test_is_positive(self, value):
        self.assertGreater(value, 0)
```

4.3.2 Mixin

```
import unittest
from generator import generate, GeneratorMixin
```

(continues on next page)

(continued from previous page)

```
class MyTestCase(GeneratorMixin, unittest.TestCase):

    @generate(1, 2, 3):
    def test_is_positive(self, value):
        self.assertGreater(value, 0)


class AnotherMyTestCase(MyTestCase):

    @generate(1, 3, 5):
    def test_is_odd(self, value):
        self.assertTrue(value % 2)
```

4.4 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:

4.4.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/kevinastone/generator/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” is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with “feature” is open to whoever wants to implement it.

Write Documentation

generator could always use more documentation, whether as part of the official generator 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/kevinastone/generator/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 :)

4.4.2 Get Started!

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

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

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

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv generator
$ cd generator/
$ 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 with tox:

```
$ flake8 generator tests
$ python setup.py test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

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.

4.4.3 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 2.6, 2.7, 3.3, and 3.4, and for PyPy. Check https://travis-ci.org/kevinastone/generator/pull_requests and make sure that the tests pass for all supported Python versions.

4.4.4 Tips

To run a subset of tests:

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

4.5 Credits

4.5.1 Development Lead

- Kevin Stone <kevinastone@gmail.com>

4.5.2 Contributors

None yet. Why not be the first?

4.6 History

4.7 0.1.2 (2016-08-16)

- Added better support for `nose-attrib` tags.

4.8 0.1.1 (2015-10-15)

- First release on PyPI.
- genindex
- modindex
- search