Fluent Test Documentation

Release 3.0.0

Dave Shawley

March 04, 2016

Contents

"When a failing test makes us read 20+ lines of test code, we die inside." - C.J. Gaconnet

CHAPTER 1

Why?

This is an attempt to make Python testing more readable while maintaining a Pythonic look and feel. As powerful and useful as the unittest module is, I've always disliked the Java-esque naming convention amongst other things.

While truly awesome, attempts to bring BDD to Python never feel *Pythonic*. Most of the frameworks that I have seen rely on duplicated information between the specification and the test cases. My belief is that we need something closer to what RSpec offers but one that feels like Python.

CHAPTER 2

Where?

- Source Code: https://github.com/dave-shawley/fluent-test
- CI: https://travis-ci.org/dave-shawley/fluent-test
- Documentation: https://fluent-test.readthedocs.org/

CHAPTER 3

How?

fluenttest.test_case.TestCase implements the Arrange, Act, Assert method of testing. The configuration for the test case and the execution of the single action under test is run precisely once per test case instance. The test case contains multiple assertions each in its own method. The implementation leverages existing test case runners such as nose and py.test. In order to run the arrange and act steps once per test, fluenttest calls `arrange` and `act` from within the `setUpClass` class method. Each assertion is then written in its own test method. The following snippet rewrites the simple example from the Python Standard library unittest documentation:

```
import random
import unittest

class TestSequenceFunctions(unittest.TestCase):
   def setUp(self):
      self.seq = list(range(10))

   def test_shuffle(self):
      # make sure the shuffled sequence does not lose any elements
      random.shuffle(self.seq)
      self.seq.sort()
      self.assertEqual(self.seq, list(range(10)))

      # should raise an exception for an immutable sequence
      self.assertRaises(TypeError, random.shuffle, (1, 2, 3))
```

This very simple test looks like the following when written using `fluenttest`. Notice that the comments in the original test really pointed out that there were multiple assertions buried in the test method. This is much more explicit with `fluenttest`:

```
import random
import unittest

from fluenttest import test_case

class WhenShufflingSequence(test_case.TestCase, unittest.TestCase):
    @classmethod
    def arrange(cls):
        super(WhenShufflingSequence, cls).arrange()
        cls.input_sequence = list(range(10))
        cls.result_sequence = cls.input_sequence[:]

    @classmethod
    def act(cls):
        random.shuffle(cls.result_sequence)
```

```
    def test_should_not_loose_elements(self):
        self.assertEqual(sorted(self.result_sequence),
                         sorted(self.input_sequence))

class WhenShufflingImmutableSequence(test_case.TestCase, unittest.TestCase):
    allowed_exceptions = TypeError

    @classmethod
    def act(cls):
        random.shuffle((1, 2, 3))

    def test_should_raise_type_error(self):
        self.assertIsInstance(self.exception, TypeError)
```

The `fluenttest` version is almost twice the length of the original so brevity is not a quality to expect from this style of testing. The first thing that you gain is that the comments that explained what each test is doing is replace with very explicit code. In this simplistic example, the gain isn't very notable. Look at the *tests* directory for a realistic example of tests written in this style.

CHAPTER 4

Contributing

Contributions are welcome as long as they follow a few basic rules:

1. They start out life by forking the central repo and creating a new branch off of *master*.
2. All tests pass and coverage is at 100% - **make test**
3. All quality checks pass - **make lint**
4. Issue a pull-request through github.

4.1 Development Environment

Like many other projects, the development environment is contained in a virtual environment and controlled by a Makefile. The inclusion of make is less than perfect, but it is the easiest way to bootstrap a project on just about any platform. Start out by cloning the repository with git and building a virtual environment to work with:

```
$ git clone https://github.com/my-org/fluent-test.git
$ cd fluent-test
$ make environment
```

This will create a Python 3 environment in the *env* directory using *mkvenv* and install the various prerequisites such as *pip* and *nose*. You can activate the environment source `source env/bin/activate`, launch a Python interpreter with `env/bin/python`, and run the test suite with `env/bin/nosetests`.

The Makefile exports a few other useful targets:

- **make test**: run the tests
- **make lint**: run various static analysis tools
- **make clean**: remove cache files
- **make mostly-clean**: remove built and cached eggs
- **make dist-clean**: remove generated distributions
- **make maintainer-clean**: remove virtual environment
- **make sdist**: create a distribution tarball
- **make docs**: build the HTML documentation

4.1.1 Unit Testing

TestCase follows the *Arrange, Act, Assert* pattern espoused by the TDD community; however, the implementation may seem confounding at first. The arrange and action steps are implemented as class methods and the assertions are implemented as instance methods. This is a side-effect of using existing test runners and little more. The goal is to assure that the arrangement and action methods are run precisely once before the assertions are checked. The common unit test running tools follow the same approach as their xUnit brethren.

1. Run class-level *setup* method(s)
2. **For each test case defined in the class, do the following:**
    (a) Run the instance-level *setup* methods(s)
    (b) Run the test function
    (c) Run the instance-level *tear down* method(s)
3. Run class-level *tear down* methods

The AAA approach to unit testing encourages a single action under test along with many atomic assertions. In an xUnit framework, it is natural to model assertions as specific tests. Each test should be a single assertion to ensure that the root of a failure is as succinct as possible. Since we want many small assertions for a single arrangement or environment, we use the class-level setup and tear down methods.

TestCase implements a class-level setup method that delegates the *arrange* and *action* steps to sub-class defined methods named `arrange` and `act`. Test case implementations should implement class methods named `arrange` and `act` then implement test cases for each assertion:

```
class TheFrobinator(TestCase):
    @classmethod
    def arrange(cls):
        super(TheFrobinator, cls).arrange()
        cls.swizzle = cls.patch('frobination.internal.swizzle')
        cls.argument = 'One'
        cls.frobinator = Frobinator()

    @classmethod
    def act(cls):
        cls.return_value = cls.frobinator.frobinate(cls.argument)

    def test_should_return_True(self):
        assert self.return_value == True

    def test_should_swizzle_the_argument(self):
        self.swizzle.assert_called_once_with(self.argument)
```

Patching

The example included an instance of creating a patch as well. Fluent Test incorporates Michael Foord's excellent mock library and exposes patching as the *TestCase.patch()* and *TestCase.patch_instance()* methods. Both methods patch out a specific target from the time that the patch method is called until the class-level tear down method is invoked. Patching is a great method for isolating the class that is under test since you can replace the collaborating classes, control their behavior, and place assertions over each of the interactions.

There are two primary use cases that *TestCase* exposes. The most common one is exposed by *TestCase.patch()*. It patches the target by calling `mock.patch()`, starts the patch, and returns the patched object. *TestCase.patch_instance()* is similar except that it is really meant for patching types. It returns a tuple of the patcher and `patcher.return_value`. This simplifies the common case of patching a class to

control/inspect the instance of the class created in the unit under test. To continue our previous example, if the `Frobinator` creates an instance of the `Swizzler`, then we can use the following to test it:

```
class TheFrobinator(TestCase):
    @classmethod
    def arrange(cls);
        super(TheFrobinator, cls).arrange()
        cls.swizzler_cls, cls.swizzler_inst = cls.patch_instance(
            'frobination.Swizzler')
        cls.argument = 'One'
        cls.frobinator = Frobinator()

    @classmethod
    def act(cls):
        cls.return_value = cls.frobinator.frobinate(cls.argument)

    def test_should_create_a_Swizzler(self):
        self.swizzler_cls.assert_called_once_with()

    def test_should_swizzle_the_arguments(self):
        self.swizzler_inst.swizzle.assert_called_once_with(self.argument)
```

Exception Handling

Another useful extension that *`TestCase`* provides is to wrap the action in a `try-except` block. The test case can list exceptions that it is interested in receiving by adding the class attribute *`allowed_exceptions`* containing a `tuple` of exception classes. When an exception is raised from *`act()`* and it is listed in `allowed_exceptions`, then it is saved in the *`exception`* for later inspection. Otherwise, it is raised and propagates outward.

4.1.2 API Reference

TestCase

class `fluenttest.test_case.`**`TestCase`**

Arrange, Act, Assert test case.

Sub-classes implement test cases by *arranging* the environment in the *`arrange()`* class method, perform the *action* in the *`act()`* class method, and implement *assertions* as test methods. The individual assertion methods have to be written in such a way that the test runner in use finds them.

`allowed_exceptions`

The exception or list of exceptions that the test case is interested in capturing. An exception raised from *`act()`* will be stored in *`exception`*.

`exception`

The exception that was thrown during the action or `None`.

classmethod `act``()`

The action to test.

Subclasses are required to replace this method.

`allowed_exceptions` **= ()**

Catch this set of exception classes.

classmethod `arrange``()`

Arrange the testing environment.

Concrete test classes will probably override this method and should invoke this implementation via `super()`.

classmethod **`destroy`**`()`
Perform post-test cleanup.

Concrete tests classes may override this method if there are actions that need to be performed after *`act()`* is called. Subclasses should invoke this implementation via `super()`.

This method is guaranteed to be called *after* the action under test is invoked and before `teardown_class()`. It will be called after any captured exception has been caught.

classmethod **`patch`**(*target*, ***kwargs*)
Patch a named class or method.

Parameters **`target`** (*`str`*) – the dotted-name to patch

Returns the result of starting the patch.

This method calls `mock.patch()` with *target* and ***kwargs*, saves the result, and returns the running patch.

classmethod **`patch_instance`**(*target*, ***kwargs*)
Patch a named class and return the created instance.

Parameters **`target`** (*`str`*) – the dotted-name of the class to patch

Returns tuple of (patched class, patched instance)

This method calls *`patch()`* with ***kwargs* to patch *target* and returns a tuple containing the patched class as well as the `return_value` attribute of the patched class. This is useful if you want to patch a class and manipulate the result of the code under test creating an instance of the class.

classmethod **`setUpClass`**`()`
Arrange the environment and perform the action.

This method ensures that *`arrange()`* and *`act()`* are invoked exactly once before the assertions are fired. If you do find the need to extend this method, you should call this implementation as the last statement in your extension method as it will perform the action under test when it is called.

classmethod **`tearDownClass`**`()`
Stop any patches that have been created.

4.1.3 Change Log

Version 3.0.0

- Remove class based testing module.

  After using this library for a while, it has become apparent that the class-based testing isn't useful. It also is rather un-pythonic to assert structural type information. If you are using this, then feel free to copy and paste the code from the previous version.

- Remove top-level exports from package __init__.py.

  If you were referencing the test case as `fluenttest.TestCase`, I apologize. Removing the top level `import` statements makes it possible to reach into the version information without loading the package and its dependencies.

- Switch to Python `unittest` naming conventions.

Using `setup_class` and `teardown_class` causes problems if you run tests with `unittest.main`. Not to mention that the Standard Library uses `setUpClass` and `tearDownClass` regardless of how un-pythonic the names are.

Version 2.0.1 (15-Feb-2014)

- Correct a packaging version defect.

  Setup.py cannot safely retrieve the version from the `__version__` attribute of the package since the import requires `mock` to be present. The immediate hot-fix is to duplicate the version number until I can come up with a cleaner solution.

Version 2.0.0 (15-Feb-2014)

- Remove `fluenttest.TestCase.patches` attribute.

  The `patches` attribute was just a little too magical for my tastes and it wasn't really necessary. Removing this attribute also removed the `patch_name` parameter to `patch`. The latter change actually simplifies things quite a bit since we no longer have to derive safe attribute names.

- Add `fluenttest.TestCase.destroy()`
- Switch to semantic versioning
- Expose library version with `__version__` attribute
- Add Makefile to simplify development process
- Remove usage of tox

Version 1 (27-Jul-2013)

- Implements `fluenttest.TestCase`
- Implements `fluenttest.ClassTester`

A

D

E

P

S

T