unittest: An under-appreciated gem

Andrew Bennetts

3rd December, 2008

CANONICAL

What I hope you get out of this talk

In no particular order. . .

What I hope you get out of this talk

In no particular order...

- Why I think Python's `unittest` module is good

CANONICAL

What I hope you get out of this talk

In no particular order...

- Why I think Python's `unittest` module is good
- A new appreciation of what's possible with Python's unittest module

CANONICAL

What I hope you get out of this talk

In no particular order...

- Why I think Python's `unittest` module is good
- A new appreciation of what's possible with Python's unittest module
- Some general insights into good unit tests

CANONICAL

What I hope you get out of this talk

In no particular order...

- Why I think Python's `unittest` module is good
- A new appreciation of what's possible with Python's unittest module
- Some general insights into good unit tests

CANONICAL

Disclaimer

I'm **not** doing a comparison with other test frameworks/tools (Nose, py.test, doctest, *insert your favorite here*, . . .)

What makes a good unit test suite?

What makes a good unit test suite?

Readable: intent of each test is clear, implementation is clear

Reliable: only fails when it should, only passes when it should

Usable: easy to run, fast to run, easy to debug (if necessary!)

CANONICAL

unittest basics

Here's a *very* quick overview of unittest.

unittest basics — toy example

```
class TestFrobnicator(unittest.TestCase):
    def setUp(self):
        self.frobnicator = Frobnicator()
        self.frobnicator.initialise()

    def test_frob_one_word(self):
        input = "word"
        output = self.frobnicator.frob(input)
        self.assertEqual("frob", output)

    def test_frob_two_words(self):
        input = "two words"
        output = self.frobnicator.frob(input)
        self.assertEqual("frob frob", output)
```

CANONICAL

unittest basics — toy example

```
class TestFrobnicator(unittest.TestCase):
    def setUp(self):
        self.frobnicator = Frobnicator()
        self.frobnicator.initialise()

    def test_frob_one_word(self):
        input = "word"
        output = self.frobnicator.frob(input)
        self.assertEqual("frob", output)

    def test_frob_two_words(self):
        input = "two words"
        output = self.frobnicator.frob(input)
        self.assertEqual("frob frob", output)
```

CANONICAL

unittest basics — toy example

```
class TestFrobnicator(unittest.TestCase):
    def setUp(self):
        self.frobnicator = Frobnicator()
        self.frobnicator.initialise()

    def test_frob_one_word(self):
        input = "word"
        output = self.frobnicator.frob(input)
        self.assertEqual("frob", output)

    def test_frob_two_words(self):
        input = "two words"
        output = self.frobnicator.frob(input)
        self.assertEqual("frob frob", output)
```

CANONICAL

unittest basics — toy example

```
class TestFrobnicator(unittest.TestCase):
    def setUp(self):
        self.frobnicator = Frobnicator()
        self.frobnicator.initialise()

    def test_frob_one_word(self):
        input = "word"
        output = self.frobnicator.frob(input)
        self.assertEqual("frob", output)

    def test_frob_two_words(self):
        input = "two words"
        output = self.frobnicator.frob(input)
        self.assertEqual("frob frob", output)
```

CANONICAL

unittest basics — TestCase

Key fact: A single unit test is represented by a TestCase instance.

unittest basics — TestCase

Key fact: A single unit test is represented by a TestCase instance.

TestCase instances have a **run** method that will:

- run **setUp**
- run the test
- run **tearDown**
- report the outcome to a TestResult object

They also provide assertion methods like **assertEquals**.

CANONICAL

unittest basics — runners, loaders, results

Other major components:

TestResult: object that can record details of a success or failure.

TestLoader: turns test methods in TestCase subclasses into TestCase instances.

TestRunner: glues everything together.

Why unittest is good

It's the Standard

It's the Standard

It's in the standard library.

- It's always available.
- Most other test frameworks interoperate with it.
- Practically every Python programmer is at least minimally familiar with it.

CANONICAL

It's the Standard

It's an implementation of *xUnit*.

- Proven design.
- Practically every non-Python programmer is at least minimally familiar with it.

CANONICAL

Structure

Structure — Isolation

When run, each test has its own TestCase *instance*.

So by default, tests are *isolated* from each other.

Structure — Reusable fixture definition

Each TestCase has a `setUp` and `tearDown` method.

This makes it easy to reuse a test fixture definition between multiple tests.

CANONICAL

Structure — More code reuse

TestCases naturally group tests with common needs.

Structure — More code reuse

TestCases naturally group tests with common needs.

So any domain-specific test helpers you add (e.g. an `assertUserHasPermission` method) have a natural home.

setUp and tearDown methods are built-in examples of this.

Structure — Naming

Tests have explicit names.

This allows clear reporting of exactly which tests are failing, and a way to run individual tests rather than the whole suite.

CANONICAL

Extensibility

Extensibility

unittest is pretty easy to extend.

Extensibility

unittest is pretty easy to extend.

(And because unittest is the standard, your unittest-compatible extensions have an ok chance of working with other frameworks.)

Extending unittest

Introduction

Here's some real world unittest extensions.

addCleanup

addCleanup is a robust way to arrange for a cleanup function to be called before tearDown. This is a powerful alternative to putting cleanup logic in a try/finally block or tearDown method. For example:

```
def test_foo(self):
    foo.lock()
    self.addCleanup(foo.unlock)
    # etc...
```

CANONICAL

requireFeature

Bazaar has tests for how it handles symlinks, but Windows doesn't support symlinks. Bazaar extended the standard TestCase class to allow a test to do:

```
class TestFileRenaming(TestCase):

    _test_needs_features = [SymlinkFeature]

    ...
```

requireFeature

Alternatively, individual test methods can call

self.requireFeature(SymlinkFeature)

requireFeature

A feature is easy to define:

```
class _SymlinkFeature(Feature):

    def _probe(self):
        return hasattr(os, 'symlink')

    def feature_name(self):
        return 'symlinks'

SymlinkFeature = _SymlinkFeature()
```

requireFeature

A feature is easy to define:

```
class _SymlinkFeature(Feature):

    def _probe(self):
        return hasattr(os, 'symlink')

    def feature_name(self):
        return 'symlinks'

SymlinkFeature = _SymlinkFeature()
```

CANONICAL

Test parameterisation

Often a test is applicable to multiple scenarios.

multiply_test_suite_by_scenarios is a function that takes a test suite and list of *scenarios*.

CANONICAL

Test parameterisation — example

Simplified test case example based on a real test case from Twisted:

```
def load_tests(standard_tests, module, loader):
    tests = testtools.multiply_test_suite_by_scenarios(
        standard_tests,
        Scenario('LineReceiver',
                 lineReceiverClass=LineReceiver),
        Scenario('LineOnlyReceiver',
                 lineReceiverClass=LineOnlyReceiver))
    return unittest.TestSuite(tests)
```

CANONICAL

Test parameterisation — example

```
class LineReceiverTests(TestCase):
    def setUp(self):
        self.lineReceiver = self.makeLineReceiver(
            self.scenario.lineReceiverClass)

    def testLongLine(self):
        self.lineReceiver.MAX_LENGTH = 5
        self.lineReceiver.dataReceived('123456\n789\n')
        ...
```

Test parameterisation — example

That suite will contain two tests built from testLongLine:

- LineReceiverTests.testLongLine(LineReceiver)
- LineReceiverTests.testLongLine(LineOnlyReceiver)

CANONICAL

Custom test loaders

Many projects do this. e.g. in Bazaar modules can provide a **load_tests** function that can return a customised TestSuite.
For example, to return a suite that runs every test twice, you could do:

```
def load_tests(standard_tests, module, loader):
    result = loader.suiteClass()
    for test in testtools.iter_suite_tests(standard_tests):
        result.addTests([test, test])
    return result
```

CANONICAL

Custom test loaders

Many projects do this. e.g. in Bazaar modules can provide a **load_tests** function that can return a customised TestSuite. For example, to return a suite that runs every test twice, you could do:

```
def load_tests(standard_tests, module, loader):
    result = loader.suiteClass()
    for test in testtools.iter_suite_tests(standard_tests):
        result.addTests([test, test])
    return result
```

CANONICAL

Some Libraries

Just quickly, a couple of libraries worth knowing about.

Testtools

Miscellaneous extensions to unittest extracted from test suites of Twisted, Bazaar, etc.

Maintained by Jonathan Lange — he's here at OSDC, so find him and say hello. And give him patches to make it even better!

https://launchpad.net/testtools

CANONICAL

SubUnit

SubUnit is a library for running unit tests in separate processes to support test isolation.

Includes an **IsolatedTestCase** class that spawns a subprocess to run the test method.

https://launchpad.net/subunit

CANONICAL

testresources

testresources is a library to manage the initialisation and lifetime of expensive test fixtures.

e.g. databases used by a test suite often only need to be constructed once but standard test isolation causes them to be constructed for every fixture. testresources can manage that resource for you.

https://launchpad.net/testresources

CANONICAL

Wrapping up

Some bits of unittest suck

Some bits of unittest suck

- No standard tool for loading and invoking a test suite.

CANONICAL

Some bits of unittest suck

- No standard tool for loading and invoking a test suite.
- Parts of the API could be better.

CANONICAL

Some bits of unittest suck

- No standard tool for loading and invoking a test suite.
- Parts of the API could be better.
- The set of built-in assertions is a bit small.

CANONICAL

Some bits of unittest suck

- No standard tool for loading and invoking a test suite.
- Parts of the API could be better.
- The set of built-in assertions is a bit small.
- The documentation of it that comes with Python could be better.

CANONICAL

But it doesn't suck badly

unittest is fundamentally quite capable.

Its shortcomings are pretty easy address with extensions...

CANONICAL

But it doesn't suck badly

unittest is fundamentally quite capable.

Its shortcomings are pretty easy address with extensions...

It would be great to get some of these extensions into standard unittest!

CANONICAL

Why I like unittest

It's standard.

It encourages good unit test structure (IMO).

It's flexible enough to let me do what I need.

CANONICAL

Things to remember

unittest is in the Python standard library, and it's actually pretty good!

There's a bunch of excellent extensions you should know about if you are writing unittest code:

- PyUnit friends: https://launchpad.net/pyunit-friends
- Testtools: https://launchpad.net/testtools

CANONICAL

Questions?

Questions?

CANONICAL