

New *and* Improved

Coming changes to unittest in Python 2.7 & 3.2

Michael Foord
michael@voidspace.org.uk
www.voidspace.org.uk
@voidspace

A lolcat free presentation.

New *and* improved: Coming changes to unittest

- [Introduction](#)
- [unittest is changing](#)
- [Evolution not revolution](#)
- [New Assert Methods](#)
- [Even More...](#)
- [assertAlmostEqual delta keyword](#)
- [Deprecations](#)
- [Deprecated Methods](#)
- [Type Specific Equality Functions](#)
- [Set Comparison](#)
- [Unicode String Comparison](#)
- [Add New type specific functions](#)
- [assertRaises](#)
- [Command Line Behaviour](#)
- [Test Discovery](#)
- [More command line options](#)
- [load_tests](#)
- [Cleanup Functions with addCleanup](#)
- [Test Skipping](#)
- [More Skipping](#)
- [As class decorator](#)
- [setUpClass and tearDownClass](#)
- [setUpModule and tearDownModule](#)
- [Minor Changes](#)
- [The unittest2 Package](#)
- [The Future](#)
- [Any Questions?](#)

Introduction

`unittest` is the Python standard library testing framework. It is sometimes known as `PyUnit` and has a rich heritage as part of the `xUnit` family of testing libraries.

Python has the best testing infrastructure available of any of the major programming languages.

`unittest` is the most widely used Python testing framework.

unittest is changing



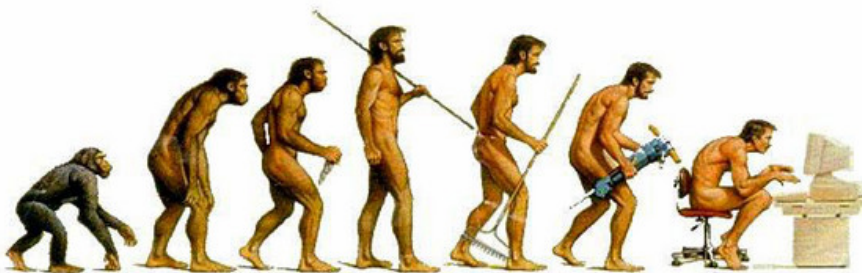
New features documented at docs.python.org/dev/library/unittest.html

Until sometime after Python 2.6 was released `unittest` was stable to the point of rigor mortis, but several developers have been working on adding much needed features and some of the most successful concepts (like test discovery) from the other major Python test frameworks. These changes will arrive in Python 2.7 and 3.2 (although a *few* of them made it into Python 3.1).

I started maintaining `unittest` about a year ago and haven't been fired yet.

Find the new stuff by looking for the *new in Python 2.7* or *changed in Python 2.7* notes in the development docs.

Evolution not revolution



Backwards compatibility is important.

No mass change to PEP8 compliant API for example. (Even PEP8 says that internal consistency is more important.)

New Assert Methods

- `assertGreater / assertLess / assertGreaterEqual / assertLessEqual`
- `assertRegexpMatches(text, regexp)` - verifies that regexp search matches text
- `assertNotRegexpMatches(text, regexp)`
- `assertIn(value, sequence) / assertNotIn` - assert membership in a container
- `assertIs(first, second) / assertIsNot` - assert identity
- `assertIsNone / assertIsNotNone`

The point of assertion methods in unittest is to provide useful messages on failure and to provide ready made methods for common assertions. Many of these were contributed by google or are in common use in other unittest extensions.

Even More...

- `assertIsInstance / assertNotIsInstance`
- `assertDictContainsSubset(subset, full)` - tests whether the key/value pairs in dictionary are a subset of those in full
- `assertSequenceEqual(actual, expected)` - ignores type of container but checks members are the same
- `assertItemsEqual(actual, expected)` - equivalent of `assertEqual(sorted(first), sorted(second))`, but works with unorderable types

`assertItemsEqual` is particularly useful in Python 3 where you can't compare (and therefore sort) objects of different types. This is still true for *some* types in Python 2 (complex for example) and it is a useful shorthand for `assertEqual(sorted(first), sorted(second))`.

assertAlmostEqual delta keyword

```
self.assertNotAlmostEqual(3.0, 3.2, delta=0.1)
```

```
import datetime
```

```
delta = datetime.timedelta(seconds=10)
second_timestamp = datetime.datetime.now()
```

```
self.assertAlmostEqual(first_timestamp,
                        second_timestamp,
                        delta=delta)
```

As well as the new methods a delta keyword argument has been added to the `assertAlmostEqual / assertNotAlmostEqual` methods. I really like this change because the default implementation of `assertAlmostEqual` is never (almost) useful to me. By default these methods round to a specified number of decimal places. When you use the delta keyword the assertion is that the difference between the two values you provide is less than (or equal to) the delta value. This permits them to be used with non-numeric values:

Deprecations



`unittest` used to have lots of ways of spelling the same methods. The duplicates have now been deprecated (but not removed).

Deprecated Methods

- `assert_` -> use `assertTrue` instead
- `fail*` -> use `assert*` instead
- `assertEquals` -> `assertEqual` is the one true way

New assertion methods *don't* have a `fail...` alias as well. If you preferred the `fail*` variant, tough luck.

Not all the 'deprecated' methods issue a `PendingDeprecationWarning` when used. `assertEquals` and `assert_` are too widely used for official deprecations, but they're deprecated in the documentation.

In the next version of the documentation the deprecated methods will be expunged and relegated to a 'deprecated methods' section.

Methods that have deprecation warnings are:

```
failUnlessEqual, failIfEqual, failUnlessAlmostEqual, failIfAlmostEqual, failUnless,
failUnlessRaises, failIf
```

Type Specific Equality Functions

More useful failure messages when comparing specific types. Used by `assertEqual` when when comparing known types:

- `assertMultilineEqual` - uses `diff`, default for comparing unicode strings
- `assertSetEqual` - default for comparing sets
- `assertDictEqual` - you get the idea
- `assertListEqual`
- `assertTupleEqual`

Set Comparison

```
Terminal — bash — 81x20
bigmac:tests michael$
bigmac:tests michael$
bigmac:tests michael$ unit2 -v test_set
testSet (test_set.TestSet) ... FAIL

=====
FAIL: testSet (test_set.TestSet)
-----
Traceback (most recent call last):
  File "/Volumes/Second Drive/repository/Presentation/Talk/test_set.py", line 11,
  in testSet
    self.assertEqual(set1, set2)
AssertionError: Items in the second set but not the first:
4

-----
Ran 1 test in 0.000s

FAILED (failures=1)
bigmac:tests michael$
```

Unicode String Comparison

```
Terminal — bash — 83x24
testStrings (test_multiline.TestMultiline) ... FAIL

=====
FAIL: testStrings (test_multiline.TestMultiline)
-----
Traceback (most recent call last):
  File "/Volumes/Second Drive/repository/Presentation/Talk/test_multiline.py", line
  15, in testStrings
    self.assertEqual(string1, string2)
AssertionError:

- A testcase is created by subclassing :class:`unittest.TestCase`. The three
+ A testcase is created by subclassing :class:`unittest2.TestCase`. The three
?
+
- individual tests are defined with methods whose names
+ individual tests are defined (probably) with methods whose names
?
+
+ start with the letters ``test``.

-----
Ran 1 test in 0.003s

FAILED (failures=1)
bigmac:tests michael$
```

Add New type specific functions

- `addTypeEqualityFunc(type, function)`

Functions added will be used by default for comparing the specified type. E.g.

```
self.addTypeEqualityFunc(
    str, self.assertMultilineEqual
)
```

Useful for comparing custom types.

Functions are used when the exact type matches, it does *not* use `isinstance`.

assertRaises

```
# as context manager
with self.assertRaises(TypeError):
    add(2, '3')

# test message with a regex
msg_re = "^You shouldn't Foo a Bar$"
with self.assertRaisesRegex(foobarerror, msg_re):
    foo_the_bar()

# access the exception object
with self.assertRaises(TypeError) as cm:
    do_something()

exception = cm.exception
self.assertEqual(exception.error_code, 3)
```

Command Line Behaviour

```
python -m unittest test_module1 test_module2
python -m unittest test_module.TestClass
python -m unittest module.TestClass.test_method
```

The `unittest` module can be used from the command line to run tests from modules, classes or even individual test methods. In earlier versions it was only possible to run individual test methods and not modules or classes.

If you are running tests for a whole test module and you define a `load_tests` function, then this function will be called to create the `TestSuite` for the module. This is the `load_tests` protocol.

You can run tests with more detail (higher verbosity) by passing in the `-v` flag:

```
python -m unittest -v test_module
```

For a list of all the command line options:

```
python -m unittest -h
```

New `verbosity` and `exit` arguments to the `main()` function (useful for interactive interpreter):

```
>>> from unittest import main
>>> main(module='test_module', verbosity=2,
...      exit=False)
```

`exit` and `verbosity` parameters are new. By default `main()` calls `sys.exit()` when it has finished the test run. This is annoying if you are using it from the interactive interpreter. You can now switch that off and run tests with a higher than default verbosity (equivalent of the `-v` command line option).

The command line can also be used for test discovery, for running all of the tests in a project or just a subset.

Test Discovery

Test discovery has been missing from unittest for a long time, forcing everyone to write their own test discovery / collection system.

```
python -m unittest discover

-v, --verbose  Verbose output
-s directory  Directory to start discovery ('.' default)
-p pattern    Pattern to match test files ('test*.py' default)
-t directory  Top level directory of project (default to start directory)
```

The options can also be passed in as positional arguments.

The options can also be passed in as positional arguments, so the following two command lines are equivalent:

```
python -m unittest discover -s project_directory -p '*_test.py'
python -m unittest discover project_directory *_test.py'
```

There are a few rules for test discovery to work, these may be relaxed in the future. *For test discovery all test modules must be importable from the top level directory of the project.*

There is an implementation of *just* the test discovery (well, plus `load_tests`) to work with standard unittest. The discover module: <http://pypi.python.org/pypi/discover>

```
pip install discover
python -m discover
```

Test discovery also works with a dotted package name as well as paths: `python -m unittest discover package.tests`

More command line options

```
-f, --failfast  Stop on first failure
-c, --catch     Catch control-C
               and display results
-b, --buffer    Buffer stdout and stderr
               during test runs
```

```
>>> from unittest import main
>>> main(module='test_module', failfast=True,
...      catchbreak=True, buffer=True, exit=False)
```

There are APIs to all these features so that they can be used by test framework authors as well as from the command line.

- `-f / --failfast`

Stop the test run on the first error or failure.

- `-c / --catch`

Control-c during the test run waits for the current test to end and then reports all the results so far. A second control-c raises the normal `KeyboardInterrupt` exception.

There are a set of functions implementing this feature available to test framework writers wishing to support this control-c handling. See [Signal Handling](#) in the development documentation.

The signal handling, the `-c` command line option, should work on all CPython platforms. It doesn't work correctly on Jython or IronPython that have missing or incomplete implementations of the [signal module](#).

- `-b / --buffer`

The standard out and standard error streams are buffered during the test run. Output during a passing test is discarded. Output is echoed normally on test fail or error and is added to the failure

messages.

The command line can also be used for test discovery, for running all of the tests in a project or just a subset.

load_tests

If a test module defines a `load_tests` function it will be called to create the test suite for the module.

This example loads tests from two specific TestCases:

```
def load_tests(loader, tests, pattern):
    suite = unittest.TestSuite()
    case1 = loader.loadTestsFromTestCase(TestCase1)
    case2 = loader.loadTestsFromTestCase(TestCase2)
    suite.addTests(case1)
    suite.addTests(case2)
    return suite
```

The `tests` argument is the standard tests that would be loaded from the module by default as a `TestSuite`. If you just want to add extra tests you can just call `addTests` on this. `pattern` is only used in the `__init__.py` of test packages when loaded from test discovery. This allows the `load_tests` function to continue (and customize) test discovery into the package. In standard test modules `pattern` will be `None`.

Cleanup Functions with addCleanup

Makes `tearDown` obsolete! Push clean-up functions onto a stack, at any point including in `setUp`, `tearDown` or inside clean-up functions, and they are guaranteed to be run when the test ends (LIFO). `self.addCleanup(function, *args, **kwargs)`:

```
def test_method(self):
    temp_dir = tempfile.mkdtemp()
    self.addCleanup(shutil.rmtree, temp_dir)
    ...
```

No need for nested `try: ... finally:` blocks in tests to clean up resources.

The full signature for `addCleanup` is: `addCleanup(function, *args, **kwargs)`. Any additional args or keyword arguments will be passed into the cleanup function when it is called. It saves the need for nested `try:..finally:` blocks to undo actions performed by the test.

If `setUp()` fails, meaning that `tearDown()` is not called, then any cleanup functions added will still be called.

If you want to manually clear out the cleanup stack you can call `doCleanups()`.

Exceptions raised inside cleanup functions will cause the test to fail, but all cleanup functions will still run.

Test Skipping

Decorators that work as class or method decorators for conditionally or unconditionally skipping tests:

```
@skip("skip this test")
def test_method(self):
    ...

info = sys.version_info
@skipIf(info[2] < 5, "only Python > 2.5")
def test_method(self):
    ...

@skipUnless(info[2] < 5, "only Python < 2.5")
def test_method(self):
```

...

More Skipping

```
def test_method(self):
    self.skipTest("skip, skippety skip")

def test_method(self):
    raise SkipTest("whoops, time to skip")

@expectedFailure
def test_that_fails(self):
    self.fail('this *should* fail')
```

Ok, so `expectedFailure` isn't for skipping tests. You use it for test that are known to fail currently. If you fix the problem, so the test starts to pass, then it will be reported as an unexpected success. This will remind you to go back and remove the `expectedFailure` decorator.

Skipped tests appear in the report as 'skipped (s)', so the number of tests run will always be the same even when skipping.

As class decorator

```
# for Python >= 2.6
@skipIf(sys.platform == 'win32')
class SomeTest(TestCase)
    ...

# Python pre-2.6
class SomeTest(TestCase)
    ...

SomeTest = skipIf(sys.platform=='win32')(SomeTest)
```

setUpClass and tearDownClass

These must be implemented as class methods.

Class and module level fixtures are implemented in `TestSuite`. When the test suite encounters a test from a new class then `tearDownClass` from the previous class (if there is one) is called, followed by `setUpClass` from the new class.

Similarly if a test is from a different module from the previous test then `tearDownModule` from the previous module is run, followed by `setUpModule` from the new module.

After all the tests in the suite have run the final `tearDownClass` and `tearDownModule` are run.

```
import unittest

class Test(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls._connection = createConnection()

    @classmethod
    def tearDownClass(cls):
        cls._connection.destroy()
```

If you want the `setUpClass` and `tearDownClass` on base classes called then you must call up to them yourself. The implementations in `TestCase` are empty.

If an exception is raised during a `setUpClass` then the tests in the class are not run and the `tearDownClass` is not run. Skipped classes will not have `setUpClass` or `tearDownClass` run.

setUpModule and tearDownModule

These should be implemented as functions.

```
def setUpModule():
    createConnection()

def tearDownModule():
    closeConnection()
```

If an exception is raised in a `setUpModule` then none of the tests in the module will be run and the `tearDownModule` will not be run.

The default ordering of tests created by the unittest test loaders is to group all tests from the same modules and classes together. This will lead to `setUpClass` / `setUpModule` (etc) being called exactly once per class and module. If you randomize the order so that tests from different modules and classes are adjacent to each other then these shared fixture functions may be called multiple times.

If there are any exceptions raised during one of these functions / methods then the test is reported as an error. Because there is no corresponding test instance an `_ErrorHolder` object (that has the same interface as a `TestCase`) is created to represent the error. If you are just using the standard unittest test runner then this detail doesn't matter, but if you are a framework author it may be relevant.

Caution!

Note that shared fixtures do not play well with features like test parallelization and they also break test isolation. They should be used with care.

A `setUpModule` or `setUpClass` that raises a `SkipTest` exception will currently be reported as an error rather than a skip (although the effect is the same). This will be fixed at some point in the future.

Minor Changes

- unittest is now a package instead of a module
- Better messages with the `longMessage` class attribute
- `TestResult`: `startTestRun` and `stopTestRun`
- `TextTestResult` public and the `TextTestRunner` takes a `runnerclass` argument for providing a custom result class (you used to have to subclass `TextTestRunner` and override `_makeResult`)
- `TextTestResult` adds the test name to the test description even if you provide a docstring

The unittest2 Package

```
pip install unittest2
```

- <http://pypi.python.org/pypi/unittest2>
- Tested with Python 2.4, 2.5 & 2.6
- <http://hg.python.org/unittest2>
- In use for development of `distutils2`

Command line functionality (test discovery) provided with the `unit2` (or `unit2.py`) script.

Replace `import unittest` with `import unittest2`.

`python -m unittest ...` works in Python 2.7 even though `unittest` is a package. In Python 2.4-2.6 this doesn't work (packages can't be executed with `-m`), hence the need for `unit2`. An alternative possibility would be to turn `unittest2` back into a single module, but that is pretty horrible.

Classes in `unittest2` derive from the equivalent classes in `unittest`, so it should be possible to use the `unittest2` test running infrastructure without having to switch all your tests to using `unittest2` immediately. Similarly you can use the new assert methods on `unittest2.TestCase` with the standard `unittest` test running infrastructure. Not all of the new features in `unittest2` will work with the standard `unittest` test

loaders and runners however.

There is also the [discover module](#) if all you want is test discovery: `python -m discover` (same command line options).

The Future

- [parameterized tests](#)
- [test outcomes](#)
- The *big* issue with unittest is extensibility

Any Questions?



- Use `unittest2` and report any bugs or problems
- Make feature requests on the Python issue tracker: bugs.python.org
- Join the [Testing in Python](#) mailing list