unittest — Automated Testing Framework

Purpose: Automated testing framework

Python's `unittest` module is based on the XUnit framework design by Kent Beck and Erich Gamma. The same pattern is repeated in many other languages, including C, Perl, Java, and Smalltalk. The framework implemented by `unittest` supports fixtures, test suites, and a test runner to enable automated testing.

Basic Test Structure

Tests, as defined by `unittest`, have two parts: code to manage test dependencies (called *fixtures*), and the test itself. Individual tests are created by subclassing `TestCase` and overriding or adding appropriate methods. In the following example, the `SimplisticTest` has a single `test()` method, which would fail if `a` is ever different from `b`.

```
# unittest_simple.py

import unittest


class SimplisticTest(unittest.TestCase):

    def test(self):
        a = 'a'
        b = 'a'
        self.assertEqual(a, b)
```

Running Tests

The easiest way to run unittest tests is use the automatic discovery available through the command line interface.

```
$ python3 -m unittest unittest_simple.py

.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

This abbreviated output includes the amount of time the tests took, along with a status indicator for each test (the "." on the first line of output means that a test passed). For more detailed test results, include the `-v` option.

```
$ python3 -m unittest -v unittest_simple.py

test (unittest_simple.SimplisticTest) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Test Outcomes

Tests have 3 possible outcomes, described in the table below.

Test Case Outcomes

Outcome	Description
ok	The test passes.
FAIL	The test does not pass, and raises an AssertionError exception.
ERROR	The test raises any exception other than AssertionError.

There is no explicit way to cause a test to “pass”, so a test’s status depends on the presence (or absence) of an exception.

```
# unittest_outcomes.py

import unittest


class OutcomesTest(unittest.TestCase):

    def testPass(self):
        return

    def testFail(self):
        self.assertFalse(True)

    def testError(self):
        raise RuntimeError('Test error!')
```

When a test fails or generates an error, the traceback is included in the output.

```
$ python3 -m unittest unittest_outcomes.py

EF.
======================================================================
ERROR: testError (unittest_outcomes.OutcomesTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".../unittest_outcomes.py", line 18, in testError
    raise RuntimeError('Test error!')
RuntimeError: Test error!

======================================================================
FAIL: testFail (unittest_outcomes.OutcomesTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".../unittest_outcomes.py", line 15, in testFail
    self.assertFalse(True)
AssertionError: True is not false

----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=1, errors=1)
```

In the previous example, `testFail()` fails and the traceback shows the line with the failure code. It is up to the person reading the test output to look at the code to figure out the meaning of the failed test, though.

```
# unittest_failwithmessage.py

import unittest


class FailureMessageTest(unittest.TestCase):

    def testFail(self):
        self.assertFalse(True, 'failure message goes here')
```

To make it easier to understand the nature of a test failure, the `fail*()` and `assert*()` methods all accept an argument `msg`, which can be used to produce a more detailed error message.

```
$ python3 -m unittest -v unittest_failwithmessage.py

testFail (unittest_failwithmessage.FailureMessageTest) ... FAIL

======================================================================
FAIL: testFail (unittest_failwithmessage.FailureMessageTest)
----------------------------------------------------------------------
```

```
Traceback (most recent call last):
  File ".../unittest_failwithmessage.py", line 12, in testFail
    self.assertFalse(True, 'failure message goes here')
AssertionError: True is not false : failure message goes here

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

Asserting Truth

Most tests assert the truth of some condition. There are two different ways to write truth-checking tests, depending on the perspective of the test author and the desired outcome of the code being tested.

```
# unittest_truth.py

import unittest


class TruthTest(unittest.TestCase):

    def testAssertTrue(self):
        self.assertTrue(True)

    def testAssertFalse(self):
        self.assertFalse(False)
```

If the code produces a value which can be evaluated as true, the method `assertTrue()` should be used. If the code produces a false value, the method `assertFalse()` make more sense.

```
$ python3 -m unittest -v unittest_truth.py

testAssertFalse (unittest_truth.TruthTest) ... ok
testAssertTrue (unittest_truth.TruthTest) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Testing Equality

As a special case, `unittest` includes methods for testing the equality of two values.

```
# unittest_equality.py

import unittest


class EqualityTest(unittest.TestCase):

    def testExpectEqual(self):
        self.assertEqual(1, 3 - 2)

    def testExpectEqualFails(self):
        self.assertEqual(2, 3 - 2)

    def testExpectNotEqual(self):
        self.assertNotEqual(2, 3 - 2)

    def testExpectNotEqualFails(self):
        self.assertNotEqual(1, 3 - 2)
```

When they fail, these special test methods produce error messages including the values being compared.

```
$ python3 -m unittest -v unittest_equality.py
```

```
testExpectEqual (unittest_equality.EqualityTest) ... ok
testExpectEqualFails (unittest_equality.EqualityTest) ... FAIL
testExpectNotEqual (unittest_equality.EqualityTest) ... ok
testExpectNotEqualFails (unittest_equality.EqualityTest) ...
FAIL

======================================================================
FAIL: testExpectEqualFails (unittest_equality.EqualityTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".../unittest_equality.py", line 15, in
testExpectEqualFails
    self.assertEqual(2, 3 - 2)
AssertionError: 2 != 1

======================================================================
FAIL: testExpectNotEqualFails (unittest_equality.EqualityTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".../unittest_equality.py", line 21, in
testExpectNotEqualFails
    self.assertNotEqual(1, 3 - 2)
AssertionError: 1 == 1

----------------------------------------------------------------------
Ran 4 tests in 0.001s

FAILED (failures=2)
```

Almost Equal?

In addition to strict equality, it is possible to test for near equality of floating point numbers using `assertAlmostEqual()` and `assertNotAlmostEqual()`.

```
# unittest_almostequal.py

import unittest


class AlmostEqualTest(unittest.TestCase):

    def testEqual(self):
        self.assertEqual(1.1, 3.3 - 2.2)

    def testAlmostEqual(self):
        self.assertAlmostEqual(1.1, 3.3 - 2.2, places=1)

    def testNotAlmostEqual(self):
        self.assertNotAlmostEqual(1.1, 3.3 - 2.0, places=1)
```

The arguments are the values to be compared, and the number of decimal places to use for the test.

```
$ python3 -m unittest unittest_almostequal.py

.F.
======================================================================
FAIL: testEqual (unittest_almostequal.AlmostEqualTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".../unittest_almostequal.py", line 12, in testEqual
    self.assertEqual(1.1, 3.3 - 2.2)
AssertionError: 1.1 != 1.0999999999999996

----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=1)
```

Containers

In addition to the generic `assertEqual()` and `assertNotEqual()`, there are special methods for comparing containers like `list`, `dict`, and `set` objects.

```
# unittest_equality_container.py

import textwrap
import unittest


class ContainerEqualityTest(unittest.TestCase):

    def testCount(self):
        self.assertCountEqual(
            [1, 2, 3, 2],
            [1, 3, 2, 3],
        )

    def testDict(self):
        self.assertDictEqual(
            {'a': 1, 'b': 2},
            {'a': 1, 'b': 3},
        )

    def testList(self):
        self.assertListEqual(
            [1, 2, 3],
            [1, 3, 2],
        )

    def testMultiLineString(self):
        self.assertMultiLineEqual(
            textwrap.dedent("""
            This string
            has more than one
            line.
            """),
            textwrap.dedent("""
            This string has
            more than two
            lines.
            """),
        )

    def testSequence(self):
        self.assertSequenceEqual(
            [1, 2, 3],
            [1, 3, 2],
        )

    def testSet(self):
        self.assertSetEqual(
            set([1, 2, 3]),
            set([1, 3, 2, 4]),
        )

    def testTuple(self):
        self.assertTupleEqual(
            (1, 'a'),
            (1, 'b'),
        )
```

Each method reports inequality using a format that is meaningful for the input type, making test failures easier to understand and correct.

```
$ python3 -m unittest unittest_equality_container.py

FFFFFFF
======================================================================
FAIL: testCount
```

```
FAIL: testCount
(unittest_equality_container.ContainerEqualityTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".../unittest_equality_container.py", line 15, in
testCount
    [1, 3, 2, 3],
AssertionError: Element counts were not equal:
First has 2, Second has 1:  2
First has 1, Second has 2:  3

======================================================================
FAIL: testDict
(unittest_equality_container.ContainerEqualityTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".../unittest_equality_container.py", line 21, in
testDict
    {'a': 1, 'b': 3},
AssertionError: {'a': 1, 'b': 2} != {'a': 1, 'b': 3}
- {'a': 1, 'b': 2}
?               ^

+ {'a': 1, 'b': 3}
?               ^


======================================================================
FAIL: testList
(unittest_equality_container.ContainerEqualityTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".../unittest_equality_container.py", line 27, in
testList
    [1, 3, 2],
AssertionError: Lists differ: [1, 2, 3] != [1, 3, 2]

First differing element 1:
2
3

- [1, 2, 3]
+ [1, 3, 2]

======================================================================
FAIL: testMultiLineString
(unittest_equality_container.ContainerEqualityTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".../unittest_equality_container.py", line 41, in
testMultiLineString
    """),
AssertionError: '\nThis string\nhas more than one\nline.\n' !=
'\nThis string has\nmore than two\nlines.\n'

- This string
+ This string has
?            ++++
- has more than one
? ----           --
+ more than two
?          ++
- line.
+ lines.
?     +


======================================================================
FAIL: testSequence
(unittest_equality_container.ContainerEqualityTest)
----------------------------------------------------------------------
Traceback (most recent call last):
```

```
  File ".../unittest_equality_container.py", line 47, in
testSequence
    [1, 3, 2],
AssertionError: Sequences differ: [1, 2, 3] != [1, 3, 2]

First differing element 1:
2
3

- [1, 2, 3]
+ [1, 3, 2]

======================================================================
FAIL: testSet
(unittest_equality_container.ContainerEqualityTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".../unittest_equality_container.py", line 53, in testSet
    set([1, 3, 2, 4]),
AssertionError: Items in the second set but not the first:
4

======================================================================
FAIL: testTuple
(unittest_equality_container.ContainerEqualityTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".../unittest_equality_container.py", line 59, in
testTuple
    (1, 'b'),
AssertionError: Tuples differ: (1, 'a') != (1, 'b')

First differing element 1:
'a'
'b'

- (1, 'a')
?      ^

+ (1, 'b')
?      ^


----------------------------------------------------------------------
Ran 7 tests in 0.005s

FAILED (failures=7)
```

Use `assertIn()` to test container membership.

```
# unittest_in.py

import unittest


class ContainerMembershipTest(unittest.TestCase):

    def testDict(self):
        self.assertIn(4, {1: 'a', 2: 'b', 3: 'c'})

    def testList(self):
        self.assertIn(4, [1, 2, 3])

    def testSet(self):
        self.assertIn(4, set([1, 2, 3]))
```

Any object that supports the `in` operator or the container API can be used with `assertIn()`.

```
$ python3 -m unittest unittest_in.py

FFF
```

```
rrr
======================================================================
FAIL: testDict (unittest_in.ContainerMembershipTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".../unittest_in.py", line 12, in testDict
    self.assertIn(4, {1: 'a', 2: 'b', 3: 'c'})
AssertionError: 4 not found in {1: 'a', 2: 'b', 3: 'c'}

======================================================================
FAIL: testList (unittest_in.ContainerMembershipTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".../unittest_in.py", line 15, in testList
    self.assertIn(4, [1, 2, 3])
AssertionError: 4 not found in [1, 2, 3]

======================================================================
FAIL: testSet (unittest_in.ContainerMembershipTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".../unittest_in.py", line 18, in testSet
    self.assertIn(4, set([1, 2, 3]))
AssertionError: 4 not found in {1, 2, 3}

----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=3)
```

Testing for Exceptions

As previously mentioned, if a test raises an exception other than `AssertionError` it is treated as an error. This is very useful for uncovering mistakes while modifying code that has existing test coverage. There are circumstances, however, in which the test should verify that some code does produce an exception. For example, if an invalid value is given to an attribute of an object. In such cases, `assertRaises()` makes the code more clear than trapping the exception in the test. Compare these two tests:

```
# unittest_exception.py
```

```
import unittest


def raises_error(*args, **kwds):
    raise ValueError('Invalid value: ' + str(args) + str(kwds))


class ExceptionTest(unittest.TestCase):

    def testTrapLocally(self):
        try:
            raises_error('a', b='c')
        except ValueError:
            pass
        else:
            self.fail('Did not see ValueError')

    def testAssertRaises(self):
        self.assertRaises(
            ValueError,
            raises_error,
            'a',
            b='c',
        )
```

The results for both are the same, but the second test using `assertRaises()` is more succinct.

```
$ python3 -m unittest -v unittest_exception.py

testAssertRaises (unittest_exception.ExceptionTest) ... ok
```

```
testTrapLocally (unittest_exception.ExceptionTest) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Test Fixtures

Fixtures are outside resources needed by a test. For example, tests for one class may all need an instance of another class that provides configuration settings or another shared resource. Other test fixtures include database connections and temporary files (many people would argue that using external resources makes such tests not “unit” tests, but they are still tests and still useful).

`unittest` includes special hooks to configure and clean up any fixtures needed by tests. To establish fixtures for each individual test case, override `setUp()` on the `TestCase`. To clean them up, override `tearDown()`. To manage one set of fixtures for all instances of a test class, override the class methods `setUpClass()` and `tearDownClass()` for the `TestCase`. And to handle especially expensive setup operations for all of the tests within a module, use the module-level functions `setUpModule()` and `tearDownModule()`.

```
# unittest_fixtures.py

import random
import unittest


def setUpModule():
    print('In setUpModule()')


def tearDownModule():
    print('In tearDownModule()')


class FixturesTest(unittest.TestCase):

    @classmethod
    def setUpClass(cls):
        print('In setUpClass()')
        cls.good_range = range(1, 10)

    @classmethod
    def tearDownClass(cls):
        print('In tearDownClass()')
        del cls.good_range

    def setUp(self):
        super().setUp()
        print('\nIn setUp()')
        # Pick a number sure to be in the range. The range is
        # defined as not including the "stop" value, so make
        # sure it is not included in the set of allowed values
        # for our choice.
        self.value = random.randint(
            self.good_range.start,
            self.good_range.stop - 1,
        )

    def tearDown(self):
        print('In tearDown()')
        del self.value
        super().tearDown()

    def test1(self):
        print('In test1()')
        self.assertIn(self.value, self.good_range)

    def test2(self):
        print('In test2()')
        self.assertIn(self.value, self.good_range)
```

When this sample test is run, the order of execution of the fixture and test methods is apparent.

```
$ python3 -u -m unittest -v unittest_fixtures.py

In setUpModule()
In setUpClass()
test1 (unittest_fixtures.FixturesTest) ...
In setUp()
In test1()
In tearDown()
ok
test2 (unittest_fixtures.FixturesTest) ...
In setUp()
In test2()
In tearDown()
ok
In tearDownClass()
In tearDownModule()

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

The `tearDown` methods may not all be invoked if there are errors in the process of cleaning up fixtures. To ensure that a fixture is always released correctly, use `addCleanup()`.

```
# unittest_addcleanup.py

import random
import shutil
import tempfile
import unittest


def remove_tmpdir(dirname):
    print('In remove_tmpdir()')
    shutil.rmtree(dirname)


class FixturesTest(unittest.TestCase):

    def setUp(self):
        super().setUp()
        self.tmpdir = tempfile.mkdtemp()
        self.addCleanup(remove_tmpdir, self.tmpdir)

    def test1(self):
        print('\nIn test1()')

    def test2(self):
        print('\nIn test2()')
```

This example test creates a temporary directory and then uses `shutil` to clean it up when the test is done.

```
$ python3 -u -m unittest -v unittest_addcleanup.py

test1 (unittest_addcleanup.FixturesTest) ...
In test1()
In remove_tmpdir()
ok
test2 (unittest_addcleanup.FixturesTest) ...
In test2()
In remove_tmpdir()
ok

----------------------------------------------------------------------
Ran 2 tests in 0.002s
```

```
OK
```

Repeating Tests with Different Inputs

It is frequently useful to run the same test logic with different inputs. Rather than defining a separate test method for each small case, a common way of doing this is to use one test method containing several related assertion calls. The problem with this approach is that as soon as one assertion fails, the rest are skipped. A better solution is to use `subTest()` to create a context for a test within a test method. If the test fails, the failure is reported and the remaining tests continue.

```
# unittest_subtest.py

import unittest


class SubTest(unittest.TestCase):

    def test_combined(self):
        self.assertRegex('abc', 'a')
        self.assertRegex('abc', 'B')
        # The next assertions are not verified!
        self.assertRegex('abc', 'c')
        self.assertRegex('abc', 'd')

    def test_with_subtest(self):
        for pat in ['a', 'B', 'c', 'd']:
            with self.subTest(pattern=pat):
                self.assertRegex('abc', pat)
```

In this example, the `test_combined()` method never runs the assertions for the patterns `'c'` and `'d'`. The `test_with_subtest()` method does, and correctly reports the additional failure. Note that the test runner still considers there to only be two test cases, even though there are three failures reported.

```
$ python3 -m unittest -v unittest_subtest.py

test_combined (unittest_subtest.SubTest) ... FAIL
test_with_subtest (unittest_subtest.SubTest) ...
======================================================================
FAIL: test_combined (unittest_subtest.SubTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".../unittest_subtest.py", line 13, in test_combined
    self.assertRegex('abc', 'B')
AssertionError: Regex didn't match: 'B' not found in 'abc'

======================================================================
FAIL: test_with_subtest (unittest_subtest.SubTest) (pattern='B')
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".../unittest_subtest.py", line 21, in test_with_subtest
    self.assertRegex('abc', pat)
AssertionError: Regex didn't match: 'B' not found in 'abc'

======================================================================
FAIL: test_with_subtest (unittest_subtest.SubTest) (pattern='d')
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".../unittest_subtest.py", line 21, in test_with_subtest
    self.assertRegex('abc', pat)
AssertionError: Regex didn't match: 'd' not found in 'abc'

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=3)
```

Skipping Tests

It is frequently useful to be able to skip a test if some external condition is not met. For example, when writing tests to check

behavior of a library under a specific version of Python there is no reason to run those tests under other versions of Python. Test classes and methods can be decorated with `skip()` to always skip the tests. The decorators `skipIf()` and `skipUnless()` can be used to check a condition before skipping.

```
# unittest_skip.py

import sys
import unittest


class SkippingTest(unittest.TestCase):

    @unittest.skip('always skipped')
    def test(self):
        self.assertTrue(False)

    @unittest.skipIf(sys.version_info[0] > 2,
                     'only runs on python 2')
    def test_python2_only(self):
        self.assertTrue(False)

    @unittest.skipUnless(sys.platform == 'Darwin',
                         'only runs on macOS')
    def test_macos_only(self):
        self.assertTrue(True)

    def test_raise_skiptest(self):
        raise unittest.SkipTest('skipping via exception')
```

For complex conditions that are difficult to express in a single expression to be passed to `skipIf()` or `skipUnless()`, a test case may raise `SkipTest` directly to cause the test to be skipped.

```
$ python3 -m unittest -v unittest_skip.py

test (unittest_skip.SkippingTest) ... skipped 'always skipped'
test_macos_only (unittest_skip.SkippingTest) ... skipped 'only
runs on macOS'
test_python2_only (unittest_skip.SkippingTest) ... skipped 'only
runs on python 2'
test_raise_skiptest (unittest_skip.SkippingTest) ... skipped
'skipping via exception'

----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK (skipped=4)
```

Ignoring Failing Tests

Rather than deleting tests that are persistently broken, they can be marked with the `expectedFailure()` decorator so the failure is ignored.

```
# unittest_expectedfailure.py

import unittest


class Test(unittest.TestCase):

    @unittest.expectedFailure
    def test_never_passes(self):
        self.assertTrue(False)

    @unittest.expectedFailure
    def test_always_passes(self):
        self.assertTrue(True)
```

If a test that is expected to fail does in fact pass, that condition is treated as a special sort of failure and reported as an “unexpected success”.

.

```
$ python3 -m unittest -v unittest_expectedfailure.py

test_always_passes (unittest_expectedfailure.Test) ...
unexpected success
test_never_passes (unittest_expectedfailure.Test) ... expected
failure

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (expected failures=1, unexpected successes=1)
```

See also

- Standard library documentation for unittest
- `doctest` – An alternate means of running tests embedded in docstrings or external documentation files.
- nose – Third-party test runner with sophisticated discovery features.
- pytest – A popular third-party test runner with support for distributed execution and an alternate fixture management system.
- testrepository – Third-party test runner used by the OpenStack project, with support for parallel execution andtracking failures.

doctest — Testing Through Documentation

trace — Follow Program Flow

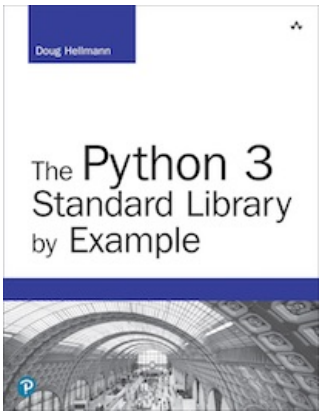
Quick Links

Basic Test Structure
Running Tests
Test Outcomes
Asserting Truth
Testing Equality
Almost Equal?
Containers
Testing for Exceptions
Test Fixtures
Repeating Tests with Different Inputs
Skipping Tests
Ignoring Failing Tests

This page was last updated 2018-03-18.

Navigation

doctest — Testing Through Documentation
trace — Follow Program Flow



Get the book

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?