26.4. `unittest` — Unit testing framework

Source code: Lib/unittest/__init__.py

(If you are already familiar with the basic concepts of testing, you might want to skip to the list of assert methods.)

The `unittest` unit testing framework was originally inspired by JUnit and has a similar flavor as major unit testing frameworks in other languages. It supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework.

To achieve this, `unittest` supports some important concepts in an object-oriented way:

test fixture
: A *test fixture* represents the preparation needed to perform one or more tests, and any associate cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.

test case
: A *test case* is the individual unit of testing. It checks for a specific response to a particular set of inputs. `unittest` provides a base class, `TestCase`, which may be used to create new test cases.

test suite
: A *test suite* is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.

test runner
: A *test runner* is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.

See also:

Module `doctest`
: Another test-support module with a very different flavor.

Simple Smalltalk Testing: With Patterns
: Kent Beck's original paper on testing frameworks using the pattern shared by `unittest`.

Nose and **py.test**
: Third-party unittest frameworks with a lighter-weight syntax for writing tests. For example, `assert func(10) == 42`.

The Python Testing Tools Taxonomy

An extensive list of Python testing tools including functional testing frameworks and mock object libraries.

Testing in Python Mailing List
A special-interest-group for discussion of testing, and testing tools, in Python.

The script `Tools/unittestgui/unittestgui.py` in the Python source distribution is a GUI tool for test discovery and execution. This is intended largely for ease of use for those new to unit testing. For production environments it is recommended that tests be driven by a continuous integration system such as Buildbot, Jenkins or Hudson.

26.4.1. Basic example

The `unittest` module provides a rich set of tools for constructing and running tests. This section demonstrates that a small subset of the tools suffice to meet the needs of most users.

Here is a short script to test three string methods:

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

A testcase is created by subclassing `unittest.TestCase`. The three individual tests are defined with methods whose names start with the letters `test`. This naming convention informs the test runner about which methods represent tests.

The crux of each test is a call to `assertEqual()` to check for an expected result; `assertTrue()` or `assertFalse()` to verify a condition; or `assertRaises()` to verify that a specific exception gets raised. These methods are used instead of the `assert` statement so the test runner can accumulate all test results and produce a report.

The `setUp()` and `tearDown()` methods allow you to define instructions that will be executed before and after each test method. They are covered in more detail in the section Organizing test code.

The final block shows a simple way to run the tests. `unittest.main()` provides a command-line interface to the test script. When run from the command line, the above script produces an output that looks like this:

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Passing the `-v` option to your test script will instruct `unittest.main()` to enable a higher level of verbosity, and produce the following output:

```
test_isupper (__main__.TestStringMethods) ... ok
test_split (__main__.TestStringMethods) ... ok
test_upper (__main__.TestStringMethods) ... ok

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

OK
```

The above examples show the most commonly used `unittest` features which are sufficient to meet many everyday testing needs. The remainder of the documentation explores the full feature set from first principles.

26.4.2. Command-Line Interface

The unittest module can be used from the command line to run tests from modules, classes or even individual test methods:

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

You can pass in a list with any combination of module names, and fully qualified class or method names.

Test modules can be specified by file path as well:

```
python -m unittest tests/test_something.py
```

This allows you to use the shell filename completion to specify the test module. The file specified must still be importable as a module. The path is converted to a module name by removing the '.py' and converting path separators into '.'. If you want to execute a test file that isn't importable as a module you should execute the file directly instead.

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

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

When executed without arguments Test Discovery is started:

```
python -m unittest
```

For a list of all the command-line options:

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

Changed in version 3.2: In earlier versions it was only possible to run individual test methods and not modules or classes.

26.4.2.1. Command-line options

unittest supports these command-line options:

-b , --buffer
: The standard output 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.

-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.

  See Signal Handling for the functions that provide this functionality.

-f , --failfast
: Stop the test run on the first error or failure.

--locals
: Show local variables in tracebacks.

New in version 3.2: The command-line options `-b`, `-c` and `-f` were added.

New in version 3.5: The command-line option `--locals`.

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

26.4.3. Test Discovery

New in version 3.2.

Unittest supports simple test discovery. In order to be compatible with test discovery, all of the test files must be modules or packages (including namespace packages) importable from the top-level directory of the project (this means that their filenames must be valid identifiers).

Test discovery is implemented in `TestLoader.discover()`, but can also be used from the command line. The basic command-line usage is:

```
cd project_directory
python -m unittest discover
```

Note: As a shortcut, `python -m unittest` is the equivalent of `python -m unittest discover`. If you want to pass arguments to test discovery the `discover` sub-command must be used explicitly.

The `discover` sub-command has the following options:

`-v`, `--verbose`
Verbose output

`-s`, `--start-directory` `directory`
Directory to start discovery (`.` default)

`-p`, `--pattern` `pattern`
Pattern to match test files (`test*.py` default)

`-t`, `--top-level-directory` `directory`
Top level directory of project (defaults to start directory)

The `-s`, `-p`, and `-t` options can be passed in as positional arguments in that order. 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"
```

As well as being a path it is possible to pass a package name, for example `myproject.subpackage.test`, as the start directory. The package name you supply will then be imported and its location on the filesystem will be used as the start directory.

Caution: Test discovery loads tests by importing them. Once test discovery has found all the test files from the start directory you specify it turns the paths into package names to import. For example `foo/bar/baz.py` will be imported as `foo.bar.baz`.

If you have a package installed globally and attempt test discovery on a different copy of the package then the import *could* happen from the wrong place. If this happens test discovery will warn you and exit.

If you supply the start directory as a package name rather than a path to a directory then discover assumes that whichever location it imports from is the location you intended, so you will not get the warning.

Test modules and packages can customize test loading and discovery by through the load_tests protocol.

Changed in version 3.4: Test discovery supports namespace packages.

26.4.4. Organizing test code

The basic building blocks of unit testing are *test cases* — single scenarios that must be set up and checked for correctness. In `unittest`, test cases are represented by `unittest.TestCase` instances. To make your own test cases you must write subclasses of `TestCase` or use `FunctionTestCase`.

The testing code of a `TestCase` instance should be entirely self contained, such that it can be run either in isolation or in arbitrary combination with any number of other test cases.

The simplest `TestCase` subclass will simply implement a test method (i.e. a method whose name starts with `test`) in order to perform specific testing code:

```
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
    def test_default_widget_size(self):
        widget = Widget('The widget')
        self.assertEqual(widget.size(), (50, 50))
```

Note that in order to test something, we use one of the `assert*()` methods provided by the `TestCase` base class. If the test fails, an exception will be raised, and `unittest` will identify the test case as a *failure*. Any other exceptions will be treated as *errors*.

Tests can be numerous, and their set-up can be repetitive. Luckily, we can factor out set-up code by implementing a method called `setUp()`, which the testing framework will automatically call for every single test we run:

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def test_default_widget_size(self):
        self.assertEqual(self.widget.size(), (50,50),
                         'incorrect default size')

    def test_widget_resize(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150),
                         'wrong size after resize')
```

Note: The order in which the various tests will be run is determined by sorting the test method names with respect to the built-in ordering for strings.

If the `setUp()` method raises an exception while the test is running, the framework will consider the test to have suffered an error, and the test method will not be executed.

Similarly, we can provide a `tearDown()` method that tidies up after the test method has been run:

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def tearDown(self):
        self.widget.dispose()
```

If `setUp()` succeeded, `tearDown()` will be run whether the test method succeeded or not.

Such a working environment for the testing code is called a *fixture*.

Test case instances are grouped together according to the features they test. `unittest` provides a mechanism for this: the *test suite*, represented by `unittest`'s `TestSuite` class. In most cases, calling `unittest.main()` will do the right thing and collect all the module's test cases for you, and then execute them.

However, should you want to customize the building of your test suite, you can do it yourself:

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('test_default_size'))
```

```
    suite.addTest(WidgetTestCase('test_resize'))
    return suite
```

You can place the definitions of test cases and test suites in the same modules as the code they are to test (such as `widget.py`), but there are several advantages to placing the test code in a separate module, such as `test_widget.py`:

- The test module can be run standalone from the command line.
- The test code can more easily be separated from shipped code.
- There is less temptation to change test code to fit the code it tests without a good reason.
- Test code should be modified much less frequently than the code it tests.
- Tested code can be refactored more easily.
- Tests for modules written in C must be in separate modules anyway, so why not be consistent?
- If the testing strategy changes, there is no need to change the source code.

26.4.5. Re-using old test code

Some users will find that they have existing test code that they would like to run from `unittest`, without converting every old test function to a `TestCase` subclass.

For this reason, `unittest` provides a `FunctionTestCase` class. This subclass of `TestCase` can be used to wrap an existing test function. Set-up and tear-down functions can also be provided.

Given the following test function:

```
def testSomething():
    something = makeSomething()
    assert something.name is not None
    # ...
```

one can create an equivalent test case instance as follows, with optional set-up and tear-down methods:

```
testcase = unittest.FunctionTestCase(testSomething,
                                     setUp=makeSomethingDB,
                                     tearDown=deleteSomethingDB)
```

Note: Even though `FunctionTestCase` can be used to quickly convert an existing test base over to a `unittest`-based system, this approach is not recommended. Taking the time to set up proper `TestCase` subclasses will make future test refactorings infinitely easier.

In some cases, the existing tests may have been written using the doctest module. If so, doctest provides a DocTestSuite class that can automatically build unittest.TestSuite instances from the existing doctest-based tests.

26.4.6. Skipping tests and expected failures

New in version 3.1.

Unittest supports skipping individual test methods and even whole classes of tests. In addition, it supports marking a test as an "expected failure," a test that is broken and will fail, but shouldn't be counted as a failure on a TestResult.

Skipping a test is simply a matter of using the skip() decorator or one of its conditional variants.

Basic skipping looks like this:

```
class MyTestCase(unittest.TestCase):

    @unittest.skip("demonstrating skipping")
    def test_nothing(self):
        self.fail("shouldn't happen")

    @unittest.skipIf(mylib.__version__ < (1, 3),
                     "not supported in this library version")
    def test_format(self):
        # Tests that work for only a certain version of the library.
        pass

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows
    def test_windows_support(self):
        # windows specific testing code
        pass
```

This is the output of running the example above in verbose mode:

```
test_format (__main__.MyTestCase) ... skipped 'not supported in this libra
test_nothing (__main__.MyTestCase) ... skipped 'demonstrating skipping'
test_windows_support (__main__.MyTestCase) ... skipped 'requires Windows'

----------------------------------------------------------------------
Ran 3 tests in 0.005s

OK (skipped=3)
```

Classes can be skipped just like methods:

```
@unittest.skip("showing class skipping")
class MySkippedTestCase(unittest.TestCase):
```

```
    def test_not_run(self):
        pass
```

`TestCase.setUp()` can also skip the test. This is useful when a resource that needs to be set up is not available.

Expected failures use the `expectedFailure()` decorator.

```
class ExpectedFailureTestCase(unittest.TestCase):
    @unittest.expectedFailure
    def test_fail(self):
        self.assertEqual(1, 0, "broken")
```

It's easy to roll your own skipping decorators by making a decorator that calls `skip()` on the test when it wants it to be skipped. This decorator skips the test unless the passed object has a certain attribute:

```
def skipUnlessHasattr(obj, attr):
    if hasattr(obj, attr):
        return lambda func: func
    return unittest.skip("{!r} doesn't have {!r}".format(obj, attr))
```

The following decorators implement test skipping and expected failures:

`@unittest.`**`skip`**(*reason*)
: Unconditionally skip the decorated test. *reason* should describe why the test is being skipped.

`@unittest.`**`skipIf`**(*condition*, *reason*)
: Skip the decorated test if *condition* is true.

`@unittest.`**`skipUnless`**(*condition*, *reason*)
: Skip the decorated test unless *condition* is true.

`@unittest.`**`expectedFailure`**
: Mark the test as an expected failure. If the test fails when run, the test is not counted as a failure.

exception `unittest.`**`SkipTest`**(*reason*)
: This exception is raised to skip a test.

  Usually you can use `TestCase.skipTest()` or one of the skipping decorators instead of raising this directly.

Skipped tests will not have `setUp()` or `tearDown()` run around them. Skipped classes will not have `setUpClass()` or `tearDownClass()` run. Skipped modules will not have `setUpModule()` or `tearDownModule()` run.

26.4.7. Distinguishing test iterations using subtests

New in version 3.4.

When some of your tests differ only by a some very small differences, for instance some parameters, unittest allows you to distinguish them inside the body of a test method using the `subTest()` context manager.

For example, the following test:

```
class NumbersTest(unittest.TestCase):

    def test_even(self):
        """
        Test that numbers between 0 and 5 are all even.
        """
        for i in range(0, 6):
            with self.subTest(i=i):
                self.assertEqual(i % 2, 0)
```

will produce the following output:

```
======================================================================
FAIL: test_even (__main__.NumbersTest) (i=1)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0

======================================================================
FAIL: test_even (__main__.NumbersTest) (i=3)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0

======================================================================
FAIL: test_even (__main__.NumbersTest) (i=5)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0
```

Without using a subtest, execution would stop after the first failure, and the error would be less easy to diagnose because the value of `i` wouldn't be displayed:

```
======================================================================
FAIL: test_even (__main__.NumbersTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0
```

26.4.8. Classes and functions

This section describes in depth the API of `unittest`.

26.4.8.1. Test cases

class `unittest.`**`TestCase`**(*methodName='runTest'*)

Instances of the `TestCase` class represent the logical test units in the `unittest` universe. This class is intended to be used as a base class, with specific tests being implemented by concrete subclasses. This class implements the interface needed by the test runner to allow it to drive the tests, and methods that the test code can use to check for and report various kinds of failure.

Each instance of `TestCase` will run a single base method: the method named *methodName*. In most uses of `TestCase`, you will neither change the *methodName* nor reimplement the default `runTest()` method.

Changed in version 3.2: `TestCase` can be instantiated successfully without providing a *methodName*. This makes it easier to experiment with `TestCase` from the interactive interpreter.

`TestCase` instances provide three groups of methods: one group used to run the test, another used by the test implementation to check conditions and report failures, and some inquiry methods allowing information about the test itself to be gathered.

Methods in the first group (running the test) are:

`setUp`()

Method called to prepare the test fixture. This is called immediately before calling the test method; other than `AssertionError` or `SkipTest`, any exception raised by this method will be considered an error rather than a test failure. The default implementation does nothing.

`tearDown`()

Method called immediately after the test method has been called and the result recorded. This is called even if the test method raised an exception, so the im-

plementation in subclasses may need to be particularly careful about checking internal state. Any exception, other than `AssertionError` or `SkipTest`, raised by this method will be considered an additional error rather than a test failure (thus increasing the total number of reported errors). This method will only be called if the `setUp()` succeeds, regardless of the outcome of the test method. The default implementation does nothing.

setUpClass()

A class method called before tests in an individual class run. `setUpClass` is called with the class as the only argument and must be decorated as a `classmethod()`:

```
@classmethod
def setUpClass(cls):
    ...
```

See Class and Module Fixtures for more details.

New in version 3.2.

tearDownClass()

A class method called after tests in an individual class have run. `tearDownClass` is called with the class as the only argument and must be decorated as a `classmethod()`:

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

See Class and Module Fixtures for more details.

New in version 3.2.

run(*result=None*)

Run the test, collecting the result into the `TestResult` object passed as *result*. If *result* is omitted or `None`, a temporary result object is created (by calling the `defaultTestResult()` method) and used. The result object is returned to `run()`'s caller.

The same effect may be had by simply calling the `TestCase` instance.

Changed in version 3.3: Previous versions of `run` did not return the result. Neither did calling an instance.

skipTest(*reason*)

Calling this during a test method or `setUp()` skips the current test. See Skipping tests and expected failures for more information.

New in version 3.1.

subTest(*msg=None*, ***params*)

Return a context manager which executes the enclosed code block as a subtest. *msg* and *params* are optional, arbitrary values which are displayed whenever a subtest fails, allowing you to identify them clearly.

A test case can contain any number of subtest declarations, and they can be arbitrarily nested.

See Distinguishing test iterations using subtests for more information.

New in version 3.4.

debug()

Run the test without collecting the result. This allows exceptions raised by the test to be propagated to the caller, and can be used to support running tests under a debugger.

The `TestCase` class provides several assert methods to check for and report failures. The following table lists the most commonly used methods (see the tables below for more assert methods):

Method	Checks that	New in
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2

All the assert methods accept a *msg* argument that, if specified, is used as the error message on failure (see also `longMessage`). Note that the *msg* keyword argument

can be passed to `assertRaises()`, `assertRaisesRegex()`, `assertWarns()`, `assertWarnsRegex()` only when they are used as a context manager.

assertEqual(*first*, *second*, *msg=None*)

Test that *first* and *second* are equal. If the values do not compare equal, the test will fail.

In addition, if *first* and *second* are the exact same type and one of list, tuple, dict, set, frozenset or str or any type that a subclass registers with `addTypeEqualityFunc()` the type-specific equality function will be called in order to generate a more useful default error message (see also the list of type-specific methods).

Changed in version 3.1: Added the automatic calling of type-specific equality function.

Changed in version 3.2: `assertMultiLineEqual()` added as the default type equality function for comparing strings.

assertNotEqual(*first*, *second*, *msg=None*)

Test that *first* and *second* are not equal. If the values do compare equal, the test will fail.

assertTrue(*expr*, *msg=None*)
assertFalse(*expr*, *msg=None*)

Test that *expr* is true (or false).

Note that this is equivalent to `bool(expr) is True` and not to `expr is True` (use `assertIs(expr, True)` for the latter). This method should also be avoided when more specific methods are available (e.g. `assertEqual(a, b)` instead of `assertTrue(a == b)`), because they provide a better error message in case of failure.

assertIs(*first*, *second*, *msg=None*)
assertIsNot(*first*, *second*, *msg=None*)

Test that *first* and *second* evaluate (or don't evaluate) to the same object.

New in version 3.1.

assertIsNone(*expr*, *msg=None*)
assertIsNotNone(*expr*, *msg=None*)

Test that *expr* is (or is not) `None`.

New in version 3.1.

assertIn(*first*, *second*, *msg=None*)

assertNotIn(*first*, *second*, *msg=None*)

Test that *first* is (or is not) in *second*.

New in version 3.1.

assertIsInstance(*obj*, *cls*, *msg=None*)
assertNotIsInstance(*obj*, *cls*, *msg=None*)

Test that *obj* is (or is not) an instance of *cls* (which can be a class or a tuple of classes, as supported by `isinstance()`). To check for the exact type, use `assertIs(type(obj), cls)`.

New in version 3.2.

It is also possible to check the production of exceptions, warnings, and log messages using the following methods:

Method	Checks that	New in
`assertRaises(exc, fun, *args, **kwds)`	`fun(*args, **kwds)` raises *exc*	
`assertRaisesRegex(exc, r, fun, *args, **kwds)`	`fun(*args, **kwds)` raises *exc* and the message matches regex *r*	3.1
`assertWarns(warn, fun, *args, **kwds)`	`fun(*args, **kwds)` raises *warn*	3.2
`assertWarnsRegex(warn, r, fun, *args, **kwds)`	`fun(*args, **kwds)` raises *warn* and the message matches regex *r*	3.2
`assertLogs(logger, level)`	The `with` block logs on *logger* with minimum *level*	3.4

assertRaises(*exception*, *callable*, **args*, ***kwds*)
assertRaises(*exception*, *msg=None*)

Test that an exception is raised when *callable* is called with any positional or keyword arguments that are also passed to `assertRaises()`. The test passes if *exception* is raised, is an error if another exception is raised, or fails if no exception is raised. To catch any of a group of exceptions, a tuple containing the exception classes may be passed as *exception*.

If only the *exception* and possibly the *msg* arguments are given, return a context manager so that the code under test can be written inline rather than as a function:

```
with self.assertRaises(SomeException):
    do_something()
```

When used as a context manager, `assertRaises()` accepts the additional keyword argument *msg*.

The context manager will store the caught exception object in its `exception` attribute. This can be useful if the intention is to perform additional checks on the exception raised:

```
with self.assertRaises(SomeException) as cm:
    do_something()

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

Changed in version 3.1: Added the ability to use `assertRaises()` as a context manager.

Changed in version 3.2: Added the `exception` attribute.

Changed in version 3.3: Added the *msg* keyword argument when used as a context manager.

assertRaisesRegex(*exception*, *regex*, *callable*, **args*, ***kwds*)
assertRaisesRegex(*exception*, *regex*, *msg=None*)

Like `assertRaises()` but also tests that *regex* matches on the string representation of the raised exception. *regex* may be a regular expression object or a string containing a regular expression suitable for use by `re.search()`. Examples:

```
self.assertRaisesRegex(ValueError, "invalid literal for.*XYZ'$",
                       int, 'XYZ')
```

or:

```
with self.assertRaisesRegex(ValueError, 'literal'):
   int('XYZ')
```

New in version 3.1: under the name `assertRaisesRegexp`.

Changed in version 3.2: Renamed to `assertRaisesRegex()`.

Changed in version 3.3: Added the *msg* keyword argument when used as a context manager.

assertWarns(*warning*, *callable*, **args*, ***kwds*)

assertWarns(*warning*, *msg=None*)

Test that a warning is triggered when *callable* is called with any positional or keyword arguments that are also passed to `assertWarns()`. The test passes if *warning* is triggered and fails if it isn't. Any exception is an error. To catch any of a group of warnings, a tuple containing the warning classes may be passed as *warnings*.

If only the *warning* and possibly the *msg* arguments are given, return a context manager so that the code under test can be written inline rather than as a function:

```
with self.assertWarns(SomeWarning):
    do_something()
```

When used as a context manager, `assertWarns()` accepts the additional keyword argument *msg*.

The context manager will store the caught warning object in its `warning` attribute, and the source line which triggered the warnings in the `filename` and `lineno` attributes. This can be useful if the intention is to perform additional checks on the warning caught:

```
with self.assertWarns(SomeWarning) as cm:
    do_something()

self.assertIn('myfile.py', cm.filename)
self.assertEqual(320, cm.lineno)
```

This method works regardless of the warning filters in place when it is called.

New in version 3.2.

Changed in version 3.3: Added the *msg* keyword argument when used as a context manager.

assertWarnsRegex(*warning*, *regex*, *callable*, **args*, ***kwds*)
assertWarnsRegex(*warning*, *regex*, *msg=None*)

Like `assertWarns()` but also tests that *regex* matches on the message of the triggered warning. *regex* may be a regular expression object or a string containing a regular expression suitable for use by `re.search()`. Example:

```
self.assertWarnsRegex(DeprecationWarning,
                      r'legacy_function\(\) is deprecated',
                      legacy_function, 'XYZ')
```

or:

```
with self.assertWarnsRegex(RuntimeWarning, 'unsafe frobnicating'):
    frobnicate('/etc/passwd')
```

New in version 3.2.

Changed in version 3.3: Added the *msg* keyword argument when used as a context manager.

assertLogs(*logger=None*, *level=None*)

A context manager to test that at least one message is logged on the *logger* or one of its children, with at least the given *level*.

If given, *logger* should be a `logging.Logger` object or a `str` giving the name of a logger. The default is the root logger, which will catch all messages.

If given, *level* should be either a numeric logging level or its string equivalent (for example either `"ERROR"` or `logging.ERROR`). The default is `logging.INFO`.

The test passes if at least one message emitted inside the `with` block matches the *logger* and *level* conditions, otherwise it fails.

The object returned by the context manager is a recording helper which keeps tracks of the matching log messages. It has two attributes:

records

A list of `logging.LogRecord` objects of the matching log messages.

output

A list of `str` objects with the formatted output of matching messages.

Example:

```
with self.assertLogs('foo', level='INFO') as cm:
   logging.getLogger('foo').info('first message')
   logging.getLogger('foo.bar').error('second message')
self.assertEqual(cm.output, ['INFO:foo:first message',
                             'ERROR:foo.bar:second message'])
```

New in version 3.4.

There are also other methods used to perform more specific checks, such as:

Method	Checks that	New in
`assertAlmostEqual(a, b)`	`round(a-b, 7) == 0`	
`assertNotAlmostEqual(a, b)`	`round(a-b, 7) != 0`	
`assertGreater(a, b)`	`a > b`	3.1

Method	Checks that	New in
`assertGreaterEqual(a, b)`	`a >= b`	3.1
`assertLess(a, b)`	`a < b`	3.1
`assertLessEqual(a, b)`	`a <= b`	3.1
`assertRegex(s, r)`	`r.search(s)`	3.1
`assertNotRegex(s, r)`	`not r.search(s)`	3.2
`assertCountEqual(a, b)`	*a* and *b* have the same elements in the same number, regardless of their order	3.2

assertAlmostEqual(*first*, *second*, *places=7*, *msg=None*, *delta=None*)
assertNotAlmostEqual(*first*, *second*, *places=7*, *msg=None*, *delta=None*)

Test that *first* and *second* are approximately (or not approximately) equal by computing the difference, rounding to the given number of decimal *places* (default 7), and comparing to zero. Note that these methods round the values to the given number of *decimal places* (i.e. like the `round()` function) and not *significant digits*.

If *delta* is supplied instead of *places* then the difference between *first* and *second* must be less or equal to (or greater than) *delta*.

Supplying both *delta* and *places* raises a `TypeError`.

Changed in version 3.2: `assertAlmostEqual()` automatically considers almost equal objects that compare equal. `assertNotAlmostEqual()` automatically fails if the objects compare equal. Added the *delta* keyword argument.

assertGreater(*first*, *second*, *msg=None*)
assertGreaterEqual(*first*, *second*, *msg=None*)
assertLess(*first*, *second*, *msg=None*)
assertLessEqual(*first*, *second*, *msg=None*)

Test that *first* is respectively >, >=, < or <= than *second* depending on the method name. If not, the test will fail:

```
>>> self.assertGreaterEqual(3, 4)
AssertionError: "3" unexpectedly not greater than or equal to "4"
```

New in version 3.1.

assertRegex(*text*, *regex*, *msg=None*)
assertNotRegex(*text*, *regex*, *msg=None*)

Test that a *regex* search matches (or does not match) *text*. In case of failure, the error message will include the pattern and the *text* (or the pattern and the part

of *text* that unexpectedly matched). *regex* may be a regular expression object or a string containing a regular expression suitable for use by `re.search()`.

New in version 3.1: under the name `assertRegexpMatches`.

Changed in version 3.2: The method `assertRegexpMatches()` has been renamed to `assertRegex()`.

New in version 3.2: `assertNotRegex()`.

New in version 3.5: The name `assertNotRegexpMatches` is a deprecated alias for `assertNotRegex()`.

assertCountEqual(*first*, *second*, *msg=None*)

Test that sequence *first* contains the same elements as *second*, regardless of their order. When they don't, an error message listing the differences between the sequences will be generated.

Duplicate elements are *not* ignored when comparing *first* and *second*. It verifies whether each element has the same count in both sequences. Equivalent to: `assertEqual(Counter(list(first)), Counter(list(second)))` but works with sequences of unhashable objects as well.

New in version 3.2.

The `assertEqual()` method dispatches the equality check for objects of the same type to different type-specific methods. These methods are already implemented for most of the built-in types, but it's also possible to register new methods using `addTypeEqualityFunc()`:

addTypeEqualityFunc(*typeobj*, *function*)

Registers a type-specific method called by `assertEqual()` to check if two objects of exactly the same *typeobj* (not subclasses) compare equal. *function* must take two positional arguments and a third msg=None keyword argument just as `assertEqual()` does. It must raise `self.failureException(msg)` when inequality between the first two parameters is detected – possibly providing useful information and explaining the inequalities in details in the error message.

New in version 3.1.

The list of type-specific methods automatically used by `assertEqual()` are summarized in the following table. Note that it's usually not necessary to invoke these methods directly.

Method	Used to compare	New in
`assertMultiLineEqual(a, b)`	strings	3.1

Method	Used to compare	New in
`assertSequenceEqual(a, b)`	sequences	3.1
`assertListEqual(a, b)`	lists	3.1
`assertTupleEqual(a, b)`	tuples	3.1
`assertSetEqual(a, b)`	sets or frozensets	3.1
`assertDictEqual(a, b)`	dicts	3.1

assertMultiLineEqual(*first*, *second*, *msg=None*)

Test that the multiline string *first* is equal to the string *second*. When not equal a diff of the two strings highlighting the differences will be included in the error message. This method is used by default when comparing strings with `assertEqual()`.

New in version 3.1.

assertSequenceEqual(*first*, *second*, *msg=None*, *seq_type=None*)

Tests that two sequences are equal. If a *seq_type* is supplied, both *first* and *second* must be instances of *seq_type* or a failure will be raised. If the sequences are different an error message is constructed that shows the difference between the two.

This method is not called directly by `assertEqual()`, but it's used to implement `assertListEqual()` and `assertTupleEqual()`.

New in version 3.1.

assertListEqual(*first*, *second*, *msg=None*)
assertTupleEqual(*first*, *second*, *msg=None*)

Tests that two lists or tuples are equal. If not, an error message is constructed that shows only the differences between the two. An error is also raised if either of the parameters are of the wrong type. These methods are used by default when comparing lists or tuples with `assertEqual()`.

New in version 3.1.

assertSetEqual(*first*, *second*, *msg=None*)

Tests that two sets are equal. If not, an error message is constructed that lists the differences between the sets. This method is used by default when comparing sets or frozensets with `assertEqual()`.

Fails if either of *first* or *second* does not have a `set.difference()` method.

New in version 3.1.

assertDictEqual(*first*, *second*, *msg=None*)

Test that two dictionaries are equal. If not, an error message is constructed that shows the differences in the dictionaries. This method will be used by default to compare dictionaries in calls to `assertEqual()`.

New in version 3.1.

Finally the `TestCase` provides the following methods and attributes:

fail(*msg=None*)

Signals a test failure unconditionally, with *msg* or `None` for the error message.

failureException

This class attribute gives the exception raised by the test method. If a test framework needs to use a specialized exception, possibly to carry additional information, it must subclass this exception in order to "play fair" with the framework. The initial value of this attribute is `AssertionError`.

longMessage

This class attribute determines what happens when a custom failure message is passed as the msg argument to an assertXYY call that fails. `True` is the default value. In this case, the custom message is appended to the end of the standard failure message. When set to `False`, the custom message replaces the standard message.

The class setting can be overridden in individual test methods by assigning an instance attribute, self.longMessage, to `True` or `False` before calling the assert methods.

The class setting gets reset before each test call.

New in version 3.1.

maxDiff

This attribute controls the maximum length of diffs output by assert methods that report diffs on failure. It defaults to 80*8 characters. Assert methods affected by this attribute are `assertSequenceEqual()` (including all the sequence comparison methods that delegate to it), `assertDictEqual()` and `assertMultiLineEqual()`.

Setting `maxDiff` to `None` means that there is no maximum length of diffs.

New in version 3.2.

Testing frameworks can use the following methods to collect information on the test:

countTestCases()

Return the number of tests represented by this test object. For `TestCase` instances, this will always be `1`.

defaultTestResult()

Return an instance of the test result class that should be used for this test case class (if no other result instance is provided to the `run()` method).

For `TestCase` instances, this will always be an instance of `TestResult`; subclasses of `TestCase` should override this as necessary.

id()

Return a string identifying the specific test case. This is usually the full name of the test method, including the module and class name.

shortDescription()

Returns a description of the test, or `None` if no description has been provided. The default implementation of this method returns the first line of the test method's docstring, if available, or `None`.

Changed in version 3.1: In 3.1 this was changed to add the test name to the short description even in the presence of a docstring. This caused compatibility issues with unittest extensions and adding the test name was moved to the `TextTestResult` in Python 3.2.

addCleanup(*function*, **args*, ***kwargs*)

Add a function to be called after `tearDown()` to cleanup resources used during the test. Functions will be called in reverse order to the order they are added (LIFO). They are called with any arguments and keyword arguments passed into `addCleanup()` when they are added.

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

New in version 3.1.

doCleanups()

This method is called unconditionally after `tearDown()`, or after `setUp()` if `setUp()` raises an exception.

It is responsible for calling all the cleanup functions added by `addCleanup()`. If you need cleanup functions to be called *prior* to `tearDown()` then you can call `doCleanups()` yourself.

`doCleanups()` pops methods off the stack of cleanup functions one at a time, so it can be called at any time.

New in version 3.1.

class `unittest.`**`FunctionTestCase`**(*testFunc*, *setUp=None*, *tearDown=None*, *description=None*)

This class implements the portion of the `TestCase` interface which allows the test runner to drive the test, but does not provide the methods which test code can use to check and report errors. This is used to create test cases using legacy test code, allowing it to be integrated into a `unittest`-based test framework.

26.4.8.1.1. Deprecated aliases

For historical reasons, some of the `TestCase` methods had one or more aliases that are now deprecated. The following table lists the correct names along with their deprecated aliases:

Method Name	Deprecated alias	Deprecated alias
`assertEqual()`	failUnlessEqual	assertEquals
`assertNotEqual()`	failIfEqual	assertNotEquals
`assertTrue()`	failUnless	assert_
`assertFalse()`	failIf	
`assertRaises()`	failUnlessRaises	
`assertAlmostEqual()`	failUnlessAlmostEqual	assertAlmostEquals
`assertNotAlmostEqual()`	failIfAlmostEqual	assertNotAlmostEquals
`assertRegex()`		assertRegexpMatches
`assertNotRegex()`		assertNotRegexpMatches
`assertRaisesRegex()`		assertRaisesRegexp

Deprecated since version 3.1: the fail* aliases listed in the second column.

Deprecated since version 3.2: the assert* aliases listed in the third column.

Deprecated since version 3.2: `assertRegexpMatches` and `assertRaisesRegexp` have been renamed to `assertRegex()` and `assertRaisesRegex()`.

Deprecated since version 3.5: the `assertNotRegexpMatches` name in favor of `assertNotRegex()`.

26.4.8.2. Grouping tests

class `unittest.`**`TestSuite`**(*tests=()*)

This class represents an aggregation of individual test cases and test suites. The class presents the interface needed by the test runner to allow it to be run as any other test case. Running a `TestSuite` instance is the same as iterating over the suite, running each test individually.

If *tests* is given, it must be an iterable of individual test cases or other test suites that will be used to build the suite initially. Additional methods are provided to add test cases and suites to the collection later on.

`TestSuite` objects behave much like `TestCase` objects, except they do not actually implement a test. Instead, they are used to aggregate tests into groups of tests that should be run together. Some additional methods are available to add tests to `TestSuite` instances:

`addTest`(*test*)

Add a `TestCase` or `TestSuite` to the suite.

`addTests`(*tests*)

Add all the tests from an iterable of `TestCase` and `TestSuite` instances to this test suite.

This is equivalent to iterating over *tests*, calling `addTest()` for each element.

`TestSuite` shares the following methods with `TestCase`:

`run`(*result*)

Run the tests associated with this suite, collecting the result into the test result object passed as *result*. Note that unlike `TestCase.run()`, `TestSuite.run()` requires the result object to be passed in.

`debug`()

Run the tests associated with this suite without collecting the result. This allows exceptions raised by the test to be propagated to the caller and can be used to support running tests under a debugger.

`countTestCases`()

Return the number of tests represented by this test object, including all individual tests and sub-suites.

__iter__()

Tests grouped by a `TestSuite` are always accessed by iteration. Subclasses can lazily provide tests by overriding `__iter__()`. Note that this method may be called several times on a single suite (for example when counting tests or comparing for equality) so the tests returned by repeated iterations before `TestSuite.run()` must be the same for each call iteration. After `TestSuite.run()`, callers should not rely on the tests returned by this method unless the caller uses a subclass that overrides `TestSuite._removeTestAtIndex()` to preserve test references.

Changed in version 3.2: In earlier versions the `TestSuite` accessed tests directly rather than through iteration, so overriding `__iter__()` wasn't sufficient for providing tests.

Changed in version 3.4: In earlier versions the `TestSuite` held references to each `TestCase` after `TestSuite.run()`. Subclasses can restore that behavior by overriding `TestSuite._removeTestAtIndex()`.

In the typical usage of a `TestSuite` object, the `run()` method is invoked by a `TestRunner` rather than by the end-user test harness.

26.4.8.3. Loading and running tests

class `unittest.`**TestLoader**

The `TestLoader` class is used to create test suites from classes and modules. Normally, there is no need to create an instance of this class; the `unittest` module provides an instance that can be shared as `unittest.defaultTestLoader`. Using a subclass or instance, however, allows customization of some configurable properties.

`TestLoader` objects have the following attributes:

errors

A list of the non-fatal errors encountered while loading tests. Not reset by the loader at any point. Fatal errors are signalled by the relevant a method raising an exception to the caller. Non-fatal errors are also indicated by a synthetic test that will raise the original error when run.

New in version 3.5.

`TestLoader` objects have the following methods:

loadTestsFromTestCase(*testCaseClass*)

Return a suite of all test cases contained in the `TestCase`-derived `testCaseClass`.

A test case instance is created for each method named by `getTestCaseNames()`. By default these are the method names beginning with `test`. If `getTestCaseNames()` returns no methods, but the `runTest()` method is implemented, a single test case is created for that method instead.

`loadTestsFromModule`(*module*, *pattern=None*)

Return a suite of all test cases contained in the given module. This method searches *module* for classes derived from `TestCase` and creates an instance of the class for each test method defined for the class.

> **Note:** While using a hierarchy of `TestCase`-derived classes can be convenient in sharing fixtures and helper functions, defining test methods on base classes that are not intended to be instantiated directly does not play well with this method. Doing so, however, can be useful when the fixtures are different and defined in subclasses.

If a module provides a `load_tests` function it will be called to load the tests. This allows modules to customize test loading. This is the load_tests protocol. The *pattern* argument is passed as the third argument to `load_tests`.

Changed in version 3.2: Support for `load_tests` added.

Changed in version 3.5: The undocumented and unofficial *use_load_tests* default argument is deprecated and ignored, although it is still accepted for backward compatibility. The method also now accepts a keyword-only argument *pattern* which is passed to `load_tests` as the third argument.

`loadTestsFromName`(*name*, *module=None*)

Return a suite of all test cases given a string specifier.

The specifier *name* is a “dotted name” that may resolve either to a module, a test case class, a test method within a test case class, a `TestSuite` instance, or a callable object which returns a `TestCase` or `TestSuite` instance. These checks are applied in the order listed here; that is, a method on a possible test case class will be picked up as “a test method within a test case class”, rather than “a callable object”.

For example, if you have a module `SampleTests` containing a `TestCase`-derived class `SampleTestCase` with three test methods (`test_one()`, `test_two()`, and `test_three()`), the specifier `'SampleTests.SampleTestCase'` would cause this method to return a suite which will run all three test methods. Using the specifier `'SampleTests.SampleTestCase.test_two'` would cause it to return a test

suite which will run only the `test_two()` test method. The specifier can refer to modules and packages which have not been imported; they will be imported as a side-effect.

The method optionally resolves *name* relative to the given *module*.

Changed in version 3.5: If an `ImportError` or `AttributeError` occurs while traversing *name* then a synthetic test that raises that error when run will be returned. These errors are included in the errors accumulated by self.errors.

loadTestsFromNames(*names*, *module=None*)

Similar to `loadTestsFromName()`, but takes a sequence of names rather than a single name. The return value is a test suite which supports all the tests defined for each name.

getTestCaseNames(*testCaseClass*)

Return a sorted sequence of method names found within *testCaseClass*; this should be a subclass of `TestCase`.

discover(*start_dir*, *pattern='test*.py'*, *top_level_dir=None*)

Find all the test modules by recursing into subdirectories from the specified start directory, and return a TestSuite object containing them. Only test files that match *pattern* will be loaded. (Using shell style pattern matching.) Only module names that are importable (i.e. are valid Python identifiers) will be loaded.

All test modules must be importable from the top level of the project. If the start directory is not the top level directory then the top level directory must be specified separately.

If importing a module fails, for example due to a syntax error, then this will be recorded as a single error and discovery will continue. If the import failure is due to `SkipTest` being raised, it will be recorded as a skip instead of an error.

If a package (a directory containing a file named `__init__.py`) is found, the package will be checked for a `load_tests` function. If this exists then it will be called `package.load_tests(loader, tests, pattern)`. Test discovery takes care to ensure that a package is only checked for tests once during an invocation, even if the load_tests function itself calls `loader.discover`.

If `load_tests` exists then discovery does *not* recurse into the package, `load_tests` is responsible for loading all tests in the package.

The pattern is deliberately not stored as a loader attribute so that packages can continue discovery themselves. *top_level_dir* is stored so `load_tests` does not need to pass this argument in to `loader.discover()`.

start_dir can be a dotted module name as well as a directory.

New in version 3.2.

Changed in version 3.4: Modules that raise `SkipTest` on import are recorded as skips, not errors. Discovery works for namespace packages. Paths are sorted before being imported so that execution order is the same even if the underlying file system's ordering is not dependent on file name.

Changed in version 3.5: Found packages are now checked for `load_tests` regardless of whether their path matches *pattern*, because it is impossible for a package name to match the default pattern.

The following attributes of a `TestLoader` can be configured either by subclassing or assignment on an instance:

testMethodPrefix

String giving the prefix of method names which will be interpreted as test methods. The default value is `'test'`.

This affects `getTestCaseNames()` and all the `loadTestsFrom*()` methods.

sortTestMethodsUsing

Function to be used to compare method names when sorting them in `getTestCaseNames()` and all the `loadTestsFrom*()` methods.

suiteClass

Callable object that constructs a test suite from a list of tests. No methods on the resulting object are needed. The default value is the `TestSuite` class.

This affects all the `loadTestsFrom*()` methods.

class `unittest.`**TestResult**

This class is used to compile information about which tests have succeeded and which have failed.

A `TestResult` object stores the results of a set of tests. The `TestCase` and `TestSuite` classes ensure that results are properly recorded; test authors do not need to worry about recording the outcome of tests.

Testing frameworks built on top of `unittest` may want access to the `TestResult` object generated by running a set of tests for reporting purposes; a `TestResult` instance is returned by the `TestRunner.run()` method for this purpose.

`TestResult` instances have the following attributes that will be of interest when inspecting the results of running a set of tests:

errors

A list containing 2-tuples of `TestCase` instances and strings holding formatted tracebacks. Each tuple represents a test which raised an unexpected exception.

failures

A list containing 2-tuples of `TestCase` instances and strings holding formatted tracebacks. Each tuple represents a test where a failure was explicitly signalled using the `TestCase.assert*()` methods.

skipped

A list containing 2-tuples of `TestCase` instances and strings holding the reason for skipping the test.

New in version 3.1.

expectedFailures

A list containing 2-tuples of `TestCase` instances and strings holding formatted tracebacks. Each tuple represents an expected failure of the test case.

unexpectedSuccesses

A list containing `TestCase` instances that were marked as expected failures, but succeeded.

shouldStop

Set to `True` when the execution of tests should stop by `stop()`.

testsRun

The total number of tests run so far.

buffer

If set to true, `sys.stdout` and `sys.stderr` will be buffered in between `startTest()` and `stopTest()` being called. Collected output will only be echoed onto the real `sys.stdout` and `sys.stderr` if the test fails or errors. Any output is also attached to the failure / error message.

New in version 3.2.

failfast

If set to true `stop()` will be called on the first failure or error, halting the test run.

New in version 3.2.

tb_locals

If set to true then local variables will be shown in tracebacks.

New in version 3.5.

wasSuccessful()

Return `True` if all tests run so far have passed, otherwise returns `False`.

Changed in version 3.4: Returns `False` if there were any `unexpectedSuccesses` from tests marked with the `expectedFailure()` decorator.

stop()

This method can be called to signal that the set of tests being run should be aborted by setting the `shouldStop` attribute to `True`. `TestRunner` objects should respect this flag and return without running any additional tests.

For example, this feature is used by the `TextTestRunner` class to stop the test framework when the user signals an interrupt from the keyboard. Interactive tools which provide `TestRunner` implementations can use this in a similar manner.

The following methods of the `TestResult` class are used to maintain the internal data structures, and may be extended in subclasses to support additional reporting requirements. This is particularly useful in building tools which support interactive reporting while tests are being run.

startTest(*test*)

Called when the test case *test* is about to be run.

stopTest(*test*)

Called after the test case *test* has been executed, regardless of the outcome.

startTestRun()

Called once before any tests are executed.

New in version 3.1.

stopTestRun()

Called once after all tests are executed.

New in version 3.1.

addError(*test*, *err*)

Called when the test case *test* raises an unexpected exception. *err* is a tuple of the form returned by `sys.exc_info()`: `(type, value, traceback)`.

The default implementation appends a tuple `(test, formatted_err)` to the instance's `errors` attribute, where *formatted_err* is a formatted traceback derived from *err*.

addFailure(*test*, *err*)

Called when the test case *test* signals a failure. *err* is a tuple of the form returned by `sys.exc_info()`: `(type, value, traceback)`.

The default implementation appends a tuple `(test, formatted_err)` to the instance's `failures` attribute, where *formatted_err* is a formatted traceback derived from *err*.

addSuccess(*test*)

Called when the test case *test* succeeds.

The default implementation does nothing.

addSkip(*test*, *reason*)

Called when the test case *test* is skipped. *reason* is the reason the test gave for skipping.

The default implementation appends a tuple `(test, reason)` to the instance's `skipped` attribute.

addExpectedFailure(*test*, *err*)

Called when the test case *test* fails, but was marked with the `expectedFailure()` decorator.

The default implementation appends a tuple `(test, formatted_err)` to the instance's `expectedFailures` attribute, where *formatted_err* is a formatted traceback derived from *err*.

addUnexpectedSuccess(*test*)

Called when the test case *test* was marked with the `expectedFailure()` decorator, but succeeded.

The default implementation appends the test to the instance's `unexpectedSuccesses` attribute.

addSubTest(*test*, *subtest*, *outcome*)

Called when a subtest finishes. *test* is the test case corresponding to the test method. *subtest* is a custom `TestCase` instance describing the subtest.

If *outcome* is `None`, the subtest succeeded. Otherwise, it failed with an exception where *outcome* is a tuple of the form returned by `sys.exc_info()`: `(type, value, traceback)`.

The default implementation does nothing when the outcome is a success, and records subtest failures as normal failures.

New in version 3.4.

class unittest.**TextTestResult**(*stream*, *descriptions*, *verbosity*)

A concrete implementation of TestResult used by the TextTestRunner.

New in version 3.2: This class was previously named _TextTestResult. The old name still exists as an alias but is deprecated.

unittest.**defaultTestLoader**

Instance of the TestLoader class intended to be shared. If no customization of the TestLoader is needed, this instance can be used instead of repeatedly creating new instances.

class unittest.**TextTestRunner**(*stream=None*, *descriptions=True*, *verbosity=1*, *failfast=False*, *buffer=False*, *resultclass=None*, *warnings=None*, ***, *tb_locals=False*)

A basic test runner implementation that outputs results to a stream. If *stream* is None, the default, sys.stderr is used as the output stream. This class has a few configurable parameters, but is essentially very simple. Graphical applications which run test suites should provide alternate implementations. Such implementations should accept **kwargs as the interface to construct runners changes when features are added to unittest.

By default this runner shows DeprecationWarning, PendingDeprecationWarning, ResourceWarning and ImportWarning even if they are ignored by default. Deprecation warnings caused by deprecated unittest methods are also special-cased and, when the warning filters are 'default' or 'always', they will appear only once per-module, in order to avoid too many warning messages. This behavior can be overridden using Python's -Wd or -Wa options (see Warning control) and leaving *warnings* to None.

Changed in version 3.2: Added the warnings argument.

Changed in version 3.2: The default stream is set to sys.stderr at instantiation time rather than import time.

Changed in version 3.5: Added the tb_locals parameter.

_makeResult()

This method returns the instance of TestResult used by run(). It is not intended to be called directly, but can be overridden in subclasses to provide a custom TestResult.

_makeResult() instantiates the class or callable passed in the TextTestRunner constructor as the resultclass argument. It defaults to TextTestResult if no

`resultclass` is provided. The result class is instantiated with the following arguments:

```
stream, descriptions, verbosity
```

run(*test*)

This method is the main public interface to the *TextTestRunner*. This method takes a `TestSuite` or `TestCase` instance. A `TestResult` is created by calling `_makeResult()` and the test(s) are run and the results printed to stdout.

`unittest.`**main**(*module='__main__'*, *defaultTest=None*, *argv=None*, *testRunner=None*, *testLoader=unittest.defaultTestLoader*, *exit=True*, *verbosity=1*, *failfast=None*, *catchbreak=None*, *buffer=None*, *warnings=None*)

A command-line program that loads a set of tests from *module* and runs them; this is primarily for making test modules conveniently executable. The simplest use for this function is to include the following line at the end of a test script:

```
if __name__ == '__main__':
    unittest.main()
```

You can run tests with more detailed information by passing in the verbosity argument:

```
if __name__ == '__main__':
    unittest.main(verbosity=2)
```

The *defaultTest* argument is either the name of a single test or an iterable of test names to run if no test names are specified via *argv*. If not specified or `None` and no test names are provided via *argv*, all tests found in *module* are run.

The *argv* argument can be a list of options passed to the program, with the first element being the program name. If not specified or `None`, the values of `sys.argv` are used.

The *testRunner* argument can either be a test runner class or an already created instance of it. By default `main` calls `sys.exit()` with an exit code indicating success or failure of the tests run.

The *testLoader* argument has to be a `TestLoader` instance, and defaults to `defaultTestLoader`.

`main` supports being used from the interactive interpreter by passing in the argument `exit=False`. This displays the result on standard output without calling `sys.exit()`:

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

The *failfast*, *catchbreak* and *buffer* parameters have the same effect as the same-name command-line options.

The *warnings* argument specifies the warning filter that should be used while running the tests. If it's not specified, it will remain `None` if a `-W` option is passed to **python** (see Warning control), otherwise it will be set to `'default'`.

Calling `main` actually returns an instance of the `TestProgram` class. This stores the result of the tests run as the `result` attribute.

Changed in version 3.1: The *exit* parameter was added.

Changed in version 3.2: The *verbosity*, *failfast*, *catchbreak*, *buffer* and *warnings* parameters were added.

Changed in version 3.4: The *defaultTest* parameter was changed to also accept an iterable of test names.

26.4.8.3.1. load_tests Protocol

New in version 3.2.

Modules or packages can customize how tests are loaded from them during normal test runs or test discovery by implementing a function called `load_tests`.

If a test module defines `load_tests` it will be called by `TestLoader.loadTestsFromModule()` with the following arguments:

```
load_tests(loader, standard_tests, pattern)
```

where *pattern* is passed straight through from `loadTestsFromModule`. It defaults to `None`.

It should return a `TestSuite`.

loader is the instance of `TestLoader` doing the loading. *standard_tests* are the tests that would be loaded by default from the module. It is common for test modules to only want to add or remove tests from the standard set of tests. The third argument is used when loading packages as part of test discovery.

A typical `load_tests` function that loads tests from a specific set of `TestCase` classes may look like:

```
test_cases = (TestCase1, TestCase2, TestCase3)

def load_tests(loader, tests, pattern):
    suite = TestSuite()
    for test_class in test_cases:
        tests = loader.loadTestsFromTestCase(test_class)
        suite.addTests(tests)
    return suite
```

If discovery is started in a directory containing a package, either from the command line or by calling `TestLoader.discover()`, then the package `__init__.py` will be checked for `load_tests`. If that function does not exist, discovery will recurse into the package as though it were just another directory. Otherwise, discovery of the package's tests will be left up to `load_tests` which is called with the following arguments:

```
load_tests(loader, standard_tests, pattern)
```

This should return a `TestSuite` representing all the tests from the package. (`standard_tests` will only contain tests collected from `__init__.py`.)

Because the pattern is passed into `load_tests` the package is free to continue (and potentially modify) test discovery. A 'do nothing' `load_tests` function for a test package would look like:

```
def load_tests(loader, standard_tests, pattern):
    # top level directory cached on loader instance
    this_dir = os.path.dirname(__file__)
    package_tests = loader.discover(start_dir=this_dir, pattern=pattern)
    standard_tests.addTests(package_tests)
    return standard_tests
```

Changed in version 3.5: Discovery no longer checks package names for matching *pattern* due to the impossibility of package names matching the default pattern.

26.4.9. Class and Module Fixtures

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 have run the final `tearDownClass` and `tearDownModule` are run.

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

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 in a single test run.

Shared fixtures are not intended to work with suites with non-standard ordering. A `BaseTestSuite` still exists for frameworks that don't want to support shared fixtures.

If there are any exceptions raised during one of the shared fixture functions 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.

26.4.9.1. setUpClass and tearDownClass

These must be implemented as class methods:

```
import unittest

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

    @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. If the exception is a `SkipTest` exception then the class will be reported as having been skipped instead of as an error.

26.4.9.2. 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. If the exception is a `SkipTest` exception then the module will be reported as having been skipped instead of as an error.

26.4.10. Signal Handling

New in version 3.2.

The `-c/--catch` command-line option to unittest, along with the `catchbreak` parameter to `unittest.main()`, provide more friendly handling of control-C during a test run. With catch break behavior enabled control-C will allow the currently running test to complete, and the test run will then end and report all the results so far. A second control-c will raise a `KeyboardInterrupt` in the usual way.

The control-c handling signal handler attempts to remain compatible with code or tests that install their own `signal.SIGINT` handler. If the `unittest` handler is called but *isn't* the installed `signal.SIGINT` handler, i.e. it has been replaced by the system under test and delegated to, then it calls the default handler. This will normally be the expected behavior by code that replaces an installed handler and delegates to it. For individual tests that need `unittest` control-c handling disabled the `removeHandler()` decorator can be used.

There are a few utility functions for framework authors to enable control-c handling functionality within test frameworks.

`unittest.`**`installHandler`**()

Install the control-c handler. When a `signal.SIGINT` is received (usually in response to the user pressing control-c) all registered results have `stop()` called.

`unittest.`**`registerResult`**(*result*)

Register a `TestResult` object for control-c handling. Registering a result stores a weak reference to it, so it doesn't prevent the result from being garbage collected.

Registering a `TestResult` object has no side-effects if control-c handling is not enabled, so test frameworks can unconditionally register all results they create independently of whether or not handling is enabled.

`unittest.`**`removeResult`**(*result*)

Remove a registered result. Once a result has been removed then `stop()` will no longer be called on that result object in response to a control-c.

unittest.**removeHandler**(*function=None*)

When called without arguments this function removes the control-c handler if it has been installed. This function can also be used as a test decorator to temporarily remove the handler whilst the test is being executed:

```
@unittest.removeHandler
def test_signal_handling(self):
    ...
```