26.3. `unittest` — Unit testing framework¶

New in version 2.1.

The Python unit testing framework, sometimes referred to as “PyUnit,” is a Python language version of JUnit, by Kent Beck and Erich Gamma. JUnit is, in turn, a Java version of Kent’s Smalltalk testing framework. Each is the de facto standard unit testing framework for its respective language.

`unittest` 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. The `unittest` module provides classes that make it easy to support these qualities for a set of tests.

To achieve this, `unittest` supports some important concepts:

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 smallest 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.

The test case and test fixture concepts are supported through the `TestCase` and `FunctionTestCase` classes; the former should be used when creating new tests, and the latter can be used when integrating existing test code with a `unittest`-driven framework. When building test fixtures using `TestCase`, the `setUp()` and `tearDown()` methods can be overridden to provide initialization and cleanup for the fixture. With `FunctionTestCase`, existing functions can be passed to the constructor for these purposes. When the test is run, the fixture initialization is run first; if it succeeds, the cleanup method is run after the test has been executed, regardless of the outcome of the test. Each instance of the `TestCase` will only be used to run a single test method, so a new fixture is created for each test.

Test suites are implemented by the `TestSuite` class. This class allows individual tests and test suites to be aggregated; when the suite is executed, all tests added directly to the suite and in “child” test suites are run.

A test runner is an object that provides a single method, `run()`, which accepts a `TestCase` or `TestSuite` object as a parameter, and returns a result object. The class `TestResult` is provided for use as the result object. `unittest` provides the `TextTestRunner` as an example test runner which reports test results on the standard error stream by default. Alternate runners can be implemented for other environments (such as graphical environments) without any need to derive from a specific class.

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`.

python-mock and minimock

Tools for creating mock test objects (objects simulating external resources).

26.3.1. 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 functions from the `random` module:

```
import random
import unittest

class TestSequenceFunctions(unittest.TestCase):

   def setUp(self):
       self.seq = range(10)

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

   def testchoice(self):
       element = random.choice(self.seq)
       self.assertTrue(element in self.seq)

   def testsample(self):
       self.assertRaises(ValueError, random.sample, self.seq, 20)
       for element in random.sample(self.seq, 5):
           self.assertTrue(element in self.seq)

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; `assert_()` to verify a condition; or `assertRaises()` to verify that an expected 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.

When a `setUp()` method is defined, the test runner will run that method prior to each test. Likewise, if a `tearDown()` method is defined, the test runner will invoke that method after each test. In the example, `setUp()` was used to create a fresh sequence for each test.

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
```

Instead of `unittest.main()`, there are other ways to run the tests with a finer level of control, less terse output, and no requirement to be run from the command line. For example, the last two lines may be replaced with:

```
suite = unittest.TestLoader().loadTestsFromTestCase(TestSequenceFunctions)
unittest.TextTestRunner(verbosity=2).run(suite)
```

Running the revised script from the interpreter or another script produces the following output:

```
testchoice (__main__.TestSequenceFunctions) ... ok
testsample (__main__.TestSequenceFunctions) ... ok
testshuffle (__main__.TestSequenceFunctions) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.110s

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.

26.3.2. 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 instances of `unittest`'s `TestCase` class. To make your own test cases you must write subclasses of `TestCase`, or use `FunctionTestCase`.

An instance of a `TestCase`-derived class is an object that can completely run a single test method, together with optional set-up and tidy-up code.

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 override the `runTest()` method in order to perform specific testing code:

```
import unittest

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

Note that in order to test something, we use the one of the `assert*()` or `fail*()` 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*. This helps you identify where the problem is: *failures* are caused by incorrect results - a 5 where you expected a 6. *Errors* are caused by incorrect code - e.g., a `TypeError` caused by an incorrect function call.

The way to run a test case will be described later. For now, note that to construct an instance of such a test case, we call its constructor without arguments:

```
testCase = DefaultWidgetSizeTestCase()
```

Now, such test cases can be numerous, and their set-up can be repetitive. In the above case, constructing a `Widget` in each of 100 Widget test case subclasses would mean unsightly duplication.

Luckily, we can factor out such set-up code by implementing a method called `setUp()`, which the testing framework will automatically call for us when we run the test:

```
import unittest

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

class DefaultWidgetSizeTestCase(SimpleWidgetTestCase):
   def runTest(self):
       self.assertEqual(self.widget.size(), (50,50),
                        'incorrect default size')

class WidgetResizeTestCase(SimpleWidgetTestCase):
   def runTest(self):
       self.widget.resize(100,150)
       self.assertEqual(self.widget.size(), (100,150),
                        'wrong size after resize')
```

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 `runTest()` method will not be executed.

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

```
import unittest

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

   def tearDown(self):
       self.widget.dispose()
       self.widget = None
```

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

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

Often, many small test cases will use the same fixture. In this case, we would end up subclassing `SimpleWidgetTestCase` into many small one-method classes such as `DefaultWidgetSizeTestCase`. This is time-consuming and discouraging, so in the same vein as JUnit, `unittest` provides a simpler mechanism:

```
import unittest

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

   def tearDown(self):
       self.widget.dispose()
       self.widget = None

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

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

Here we have not provided a `runTest()` method, but have instead provided two different test methods. Class instances will now each run one of the `test*()` methods, with `self.widget` created and destroyed separately for each instance. When creating an instance we must specify the test method it is to run. We do this by passing the method name in the constructor:

```
defaultSizeTestCase = WidgetTestCase('testDefaultSize')
resizeTestCase = WidgetTestCase('testResize')
```

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:

```
widgetTestSuite = unittest.TestSuite()
widgetTestSuite.addTest(WidgetTestCase('testDefaultSize'))
widgetTestSuite.addTest(WidgetTestCase('testResize'))
```

For the ease of running tests, as we will see later, it is a good idea to provide in each test module a callable object that returns a pre-built test suite:

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('testDefaultSize'))
    suite.addTest(WidgetTestCase('testResize'))
    return suite
```

or even:

```
def suite():
    tests = ['testDefaultSize', 'testResize']

    return unittest.TestSuite(map(WidgetTestCase, tests))
```

Since it is a common pattern to create a `TestCase` subclass with many similarly named test functions, `unittest` provides a `TestLoader` class that can be used to automate the process of creating a test suite and populating it with individual tests. For example,

```
suite = unittest.TestLoader().loadTestsFromTestCase(WidgetTestCase)
```

will create a test suite that will run `WidgetTestCase.testDefaultSize()` and `WidgetTestCase.testResize`. `TestLoader` uses the `'test'` method name prefix to identify test methods automatically.

Note that the order in which the various test cases will be run is determined by sorting the test function names with the built-in `cmp()` function.

Often it is desirable to group suites of test cases together, so as to run tests for the whole system at once. This is easy, since `TestSuite` instances can be added to a `TestSuite` just as `TestCase` instances can be added to a `TestSuite`:

```
suite1 = module1.TheTestSuite()
suite2 = module2.TheTestSuite()
alltests = unittest.TestSuite([suite1, suite2])
```

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.

26.3.3. 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:

```
testcase = unittest.FunctionTestCase(testSomething)
```

If there are additional set-up and tear-down methods that should be called as part of the test case's operation, they can also be provided like so:

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

To make migrating existing test suites easier, `unittest` supports tests raising `AssertionError` to indicate test failure. However, it is recommended that you use the explicit `TestCase.fail*()` and `TestCase.assert*()` methods instead, as future versions of `unittest` may treat `AssertionError` differently.

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.

26.3.4. Classes and functions¶

class `unittest.TestCase`([*methodName*])¶

Instances of the `TestCase` class represent the smallest testable 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 test, 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 test method: the method named *methodName*. If you remember, we had an earlier example that went something like this:

```
def suite():
   suite = unittest.TestSuite()
   suite.addTest(WidgetTestCase('testDefaultSize'))
   suite.addTest(WidgetTestCase('testResize'))
   return suite
```

Here, we create two instances of `WidgetTestCase`, each of which runs a single test.

methodName defaults to `'runTest'`.

class `unittest.FunctionTestCase`(*testFunc*[, *setUp*[, *tearDown*[, *description*]]])¶

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.

class `unittest.TestSuite`([*tests*])¶

This class represents an aggregation of individual tests 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.

class `unittest.TestLoader`¶

This class is responsible for loading tests according to various criteria and returning them wrapped in a `TestSuite`. It can load all tests within a given module or `TestCase` subclass.

class `unittest.TestResult`¶

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

`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*[, *descriptions*[, *verbosity*]]])¶
A basic test runner implementation which prints results on standard error. It has a few configurable parameters, but is essentially very simple. Graphical applications which run test suites should provide alternate implementations.

`unittest.main`([*module*[, *defaultTest*[, *argv*[, *testRunner*[, *testLoader*]]]]])¶

A command-line program that runs a set of tests; 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()
```

The *testRunner* argument can either be a test runner class or an already created instance of it.

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

New in version 2.3.

26.3.5. TestCase Objects¶

Each `TestCase` instance represents a single test, but each concrete subclass may be used to define multiple tests — the concrete class represents a single test fixture. The fixture is created and cleaned up for each test case.

`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:

`TestCase.setUp`()¶
Method called to prepare the test fixture. This is called immediately before calling the test method; any exception raised by this method will be considered an error rather than a test failure. The default implementation does nothing.

`TestCase.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 implementation in subclasses may need to be particularly careful about checking internal state. Any exception raised by this method will be considered an error rather than a test failure. This method will only be called if the `setUp()` succeeds, regardless of the outcome of the test method. The default implementation does nothing.

`TestCase.run`([*result*])¶

Run the test, collecting the result into the test result object passed as *result*. If *result* is omitted or `None`, a temporary result object is created (by calling the `defaultTestCase()` method) and used; this result object is not returned to `run()`'s caller.

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

`TestCase.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 test code can use any of the following methods to check for and report failures.

`TestCase.assert_`(*expr*[, *msg*])¶
`TestCase.failUnless`(*expr*[, *msg*])¶
`TestCase.assertTrue`(*expr*[, *msg*])¶
Signal a test failure if *expr* is false; the explanation for the error will be *msg* if given, otherwise it will be `None`.

`TestCase.assertEqual`(*first*, *second*[, *msg*])¶
`TestCase.failUnlessEqual`(*first*, *second*[, *msg*])¶
Test that *first* and *second* are equal. If the values do not compare equal, the test will fail with the explanation given by *msg*, or `None`. Note that using `failUnlessEqual()` improves upon doing the comparison as the first parameter to `failUnless()`: the default value for *msg* can be computed to include representations of both *first* and *second*.

`TestCase.assertNotEqual`(*first*, *second*[, *msg*])¶
`TestCase.failIfEqual`(*first*, *second*[, *msg*])¶
Test that *first* and *second* are not equal. If the values do compare equal, the test will fail with the explanation given by *msg*, or `None`. Note that using `failIfEqual()` improves upon doing the comparison as the first parameter to `failUnless()` is that the default value for *msg* can be computed to include representations of both *first* and *second*.

`TestCase.assertAlmostEqual`(*first*, *second*[, *places*[, *msg*]])¶
`TestCase.failUnlessAlmostEqual`(*first*, *second*[, *places*[, *msg*]])¶

Test that *first* and *second* are approximately equal by computing the difference, rounding to the given number of decimal *places* (default 7), and comparing to zero. Note that comparing a given number of decimal places is not the same as comparing a given number of significant digits. If the values do not compare equal, the test will fail with the explanation given by *msg*, or `None`.

`TestCase.assertNotAlmostEqual`(*first*, *second*[, *places*[, *msg*]])¶

`TestCase.failIfAlmostEqual`(*first*, *second*[, *places*[, *msg*]])¶

Test that *first* and *second* are not approximately equal by computing the difference, rounding to the given number of decimal *places* (default 7), and comparing to zero. Note that comparing a given number of decimal places is not the same as comparing a given number of significant digits. If the values do not compare equal, the test will fail with the explanation given by *msg*, or `None`.

`TestCase.assertRaises`(*exception*, *callable*, *...*)¶

`TestCase.failUnlessRaises`(*exception*, *callable*, *...*)¶

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*.

`TestCase.failIf`(*expr*[, *msg*])¶

`TestCase.assertFalse`(*expr*[, *msg*])¶

The inverse of the `failUnless()` method is the `failIf()` method. This signals a test failure if *expr* is true, with *msg* or `None` for the error message.

`TestCase.fail`([*msg*])¶

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

`TestCase.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`.

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

`TestCase.countTestCases`()¶

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

`TestCase.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.

`TestCase.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.

`TestCase.shortDescription`()¶

Returns a one-line 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`.

26.3.6. TestSuite Objects¶

`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:

`TestSuite.addTest`(*test*)¶

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

`TestSuite.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`:

`TestSuite.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.

`TestSuite.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.

`TestSuite.countTestCases`()¶

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

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

26.3.7. TestResult Objects¶

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:

`TestResult.errors`¶

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

Changed in version 2.2: Contains formatted tracebacks instead of `sys.exc_info()` results.

`TestResult.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.fail*()` or `TestCase.assert*()` methods.

Changed in version 2.2: Contains formatted tracebacks instead of `sys.exc_info()` results.

`TestResult.testsRun`¶
The total number of tests run so far.

`TestResult.wasSuccessful`()¶
Returns `True` if all tests run so far have passed, otherwise returns `False`.

`TestResult.stop`()¶

This method can be called to signal that the set of tests being run should be aborted by setting the `TestResult`'s `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.

`TestResult.startTest`(*test*)¶

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

The default implementation simply increments the instance's `testsRun` counter.

`TestResult.stopTest`(*test*)¶

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

The default implementation does nothing.

`TestResult.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*.

`TestResult.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*.

`TestResult.addSuccess`(*test*)¶

Called when the test case *test* succeeds.

The default implementation does nothing.

26.3.8. TestLoader Objects¶

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 methods:

`TestLoader.loadTestsFromTestCase`(*testCaseClass*)¶
Return a suite of all tests cases contained in the `TestCase`-derived `testCaseClass`.

`TestLoader.loadTestsFromModule`(*module*)¶

Return a suite of all tests 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.

Warning

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.

`TestLoader.loadTestsFromName`(*name*[, *module*])¶

Return a suite of all tests 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*.

`TestLoader.loadTestsFromNames`(*names*[, *module*])¶
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.

`TestLoader.getTestCaseNames`(*testCaseClass*)¶
Return a sorted sequence of method names found within *testCaseClass*; this should be a subclass of `TestCase`.

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

`TestLoader.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.

`TestLoader.sortTestMethodsUsing`¶
Function to be used to compare method names when sorting them in `getTestCaseNames()` and all the `loadTestsFrom*()` methods. The default value is the built-in `cmp()` function; the attribute can also be set to `None` to disable the sort.

`TestLoader.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.

Table Of Contents

Previous topic

26.2. `doctest` — Test interactive Python examples