testtools Documentation

Release VERSION

The testtools authors

May 11, 2017

Contents

testtools is a set of extensions to the Python standard library's unit testing framework. These extensions have been derived from many years of experience with unit testing in Python and come from many different sources. testtools also ports recent unittest changes all the way back to Python 2.4. The next release of testtools will change that to support versions that are maintained by the Python community instead, to allow the use of modern language features within testtools.

Contents:

CHAPTER 1

testtools: tasteful testing for Python

testtools is a set of extensions to the Python standard library's unit testing framework. These extensions have been derived from many years of experience with unit testing in Python and come from many different sources.

What better way to start than with a contrived code snippet?:

```
from testtools import TestCase
from testtools.content import Content
from testtools.content_type import UTF8_TEXT
from testtools.matchers import Equals

from myproject import SillySquareServer

class TestSillySquareServer(TestCase):

    def setUp(self):
        super(TestSillySquareServer, self).setUp()
        self.server = self.useFixture(SillySquareServer())
        self.addCleanup(self.attach_log_file)

    def attach_log_file(self):
        self.addDetail(
            'log-file',
            Content(UTF8_TEXT,
                    lambda: open(self.server.logfile, 'r').readlines()))

    def test_server_is_cool(self):
        self.assertThat(self.server.temperature, Equals("cool"))

    def test_square(self):
        self.assertThat(self.server.silly_square_of(7), Equals(49))
```

Why use testtools?

Better assertion methods

The standard assertion methods that come with unittest aren't as helpful as they could be, and there aren't quite enough of them. testtools adds `assertIn`, `assertIs`, `assertIsInstance` and their negatives.

Matchers: better than assertion methods

Of course, in any serious project you want to be able to have assertions that are specific to that project and the particular problem that it is addressing. Rather than forcing you to define your own assertion methods and maintain your own inheritance hierarchy of `TestCase` classes, testtools lets you write your own "matchers", custom predicates that can be plugged into a unit test:

```
def test_response_has_bold(self):
   # The response has bold text.
   response = self.server.getResponse()
   self.assertThat(response, HTMLContains(Tag('bold', 'b')))
```

More debugging info, when you need it

testtools makes it easy to add arbitrary data to your test result. If you want to know what's in a log file when a test fails, or what the load was on the computer when a test started, or what files were open, you can add that information with `TestCase.addDetail`, and it will appear in the test results if that test fails.

Extend unittest, but stay compatible and re-usable

testtools goes to great lengths to allow serious test authors and test *framework* authors to do whatever they like with their tests and their extensions while staying compatible with the standard library's unittest.

testtools has completely parametrized how exceptions raised in tests are mapped to `TestResult` methods and how tests are actually executed (ever wanted `tearDown` to be called regardless of whether `setUp` succeeds?)

It also provides many simple but handy utilities, like the ability to clone a test, a `MultiTestResult` object that lets many result objects get the results from one test suite, adapters to bring legacy `TestResult` objects into our new golden age.

Cross-Python compatibility

testtools gives you the very latest in unit testing technology in a way that will work with Python 2.7, 3.3, 3.4, 3.5, and pypy.

If you wish to use testtools with Python 2.4 or 2.5, then please use testtools 0.9.15.

If you wish to use testtools with Python 2.6 or 3.2, then please use testtools 1.9.0.

CHAPTER 2

testtools for test authors

If you are writing tests for a Python project and you (rather wisely) want to use testtools to do so, this is the manual for you.

We assume that you already know Python and that you know something about automated testing already.

If you are a test author of an unusually large or unusually unusual test suite, you might be interested in *testtools for framework folk*.

You might also be interested in the *testtools API docs*.

Introduction

testtools is a set of extensions to Python's standard unittest module. Writing tests with testtools is very much like writing tests with standard Python, or with Twisted's "trial", or nose, except a little bit easier and more enjoyable.

Below, we'll try to give some examples of how to use testtools in its most basic way, as well as a sort of feature-by-feature breakdown of the cool bits that you could easily miss.

The basics

Here's what a basic testtools unit tests look like:

```
from testtools import TestCase
from myproject import silly

class TestSillySquare(TestCase):
    """Tests for silly square function."""

    def test_square(self):
        # 'square' takes a number and multiplies it by itself.
        result = silly.square(7)
        self.assertEqual(result, 49)
```

```
    def test_square_bad_input(self):
        # 'square' raises a TypeError if it's given bad input, say a
        # string.
        self.assertRaises(TypeError, silly.square, "orange")
```

Here you have a class that inherits from `testtools.TestCase` and bundles together a bunch of related tests. The tests themselves are methods on that class that begin with `test_`.

Running your tests

You can run these tests in many ways. testtools provides a very basic mechanism for doing so:

```
$ python -m testtools.run exampletest
Tests running...
Ran 2 tests in 0.000s

OK
```

where 'exampletest' is a module that contains unit tests. By default, `testtools.run` will *not* recursively search the module or package for unit tests. To do this, you will need to either have the discover module installed or have Python 2.7 or later, and then run:

```
$ python -m testtools.run discover packagecontainingtests
```

For more information see the Python unittest documentation, and:

```
python -m testtools.run --help
```

which describes the options available to `testtools.run`.

As your testing needs grow and evolve, you will probably want to use a more sophisticated test runner. There are many of these for Python, and almost all of them will happily run testtools tests. In particular:

- testrepository
- Trial
- nose
- unittest2
- zope.testrunner (aka zope.testing)

From now on, we'll assume that you know how to run your tests.

Running test with Distutils

If you are using Distutils to build your Python project, you can use the testtools Distutils command to integrate testtools into your Distutils workflow:

```
from distutils.core import setup
from testtools import TestCommand
setup(name='foo',
    version='1.0',
    py_modules=['foo'],
    cmdclass={'test': TestCommand}
)
```

You can then run:

```
$ python setup.py test -m exampletest
Tests running...
Ran 2 tests in 0.000s

OK
```

For more information about the capabilities of the *TestCommand* command see:

```
$ python setup.py test --help
```

You can use the setup configuration to specify the default behavior of the *TestCommand* command.

Assertions

The core of automated testing is making assertions about the way things are, and getting a nice, helpful, informative error message when things are not as they ought to be.

All of the assertions that you can find in Python standard unittest can be found in testtools (remember, testtools extends unittest). testtools changes the behaviour of some of those assertions slightly and adds some new assertions that you will almost certainly find useful.

Improved assertRaises

`TestCase.assertRaises` returns the caught exception. This is useful for asserting more things about the exception than just the type:

```
def test_square_bad_input(self):
    # 'square' raises a TypeError if it's given bad input, say a
    # string.
    e = self.assertRaises(TypeError, silly.square, "orange")
    self.assertEqual("orange", e.bad_value)
    self.assertEqual("Cannot square 'orange', not a number.", str(e))
```

Note that this is incompatible with the `assertRaises` in unittest2 and Python2.7.

ExpectedException

If you are using a version of Python that supports the `with` context manager syntax, you might prefer to use that syntax to ensure that code raises particular errors. `ExpectedException` does just that. For example:

```
def test_square_root_bad_input_2(self):
    # 'square' raises a TypeError if it's given bad input.
    with ExpectedException(TypeError, "Cannot square.*"):
        silly.square('orange')
```

The first argument to `ExpectedException` is the type of exception you expect to see raised. The second argument is optional, and can be either a regular expression or a matcher. If it is a regular expression, the `str()` of the raised exception must match the regular expression. If it is a matcher, then the raised exception object must match it. The optional third argument `msg` will cause the raised error to be annotated with that message.

assertIn, assertNotIn

These two assertions check whether a value is in a sequence and whether a value is not in a sequence. They are "assert" versions of the `in` and `not in` operators. For example:

```
def test_assert_in_example(self):
    self.assertIn('a', 'cat')
    self.assertNotIn('o', 'cat')
    self.assertIn(5, list_of_primes_under_ten)
    self.assertNotIn(12, list_of_primes_under_ten)
```

assertIs, assertIsNot

These two assertions check whether values are identical to one another. This is sometimes useful when you want to test something more strict than mere equality. For example:

```
def test_assert_is_example(self):
    foo = [None]
    foo_alias = foo
    bar = [None]
    self.assertIs(foo, foo_alias)
    self.assertIsNot(foo, bar)
    self.assertEqual(foo, bar) # They are equal, but not identical
```

assertIsInstance

As much as we love duck-typing and polymorphism, sometimes you need to check whether or not a value is of a given type. This method does that. For example:

```
def test_assert_is_instance_example(self):
    now = datetime.now()
    self.assertIsInstance(now, datetime)
```

Note that there is no `assertIsNotInstance` in testtools currently.

expectFailure

Sometimes it's useful to write tests that fail. For example, you might want to turn a bug report into a unit test, but you don't know how to fix the bug yet. Or perhaps you want to document a known, temporary deficiency in a dependency.

testtools gives you the `TestCase.expectFailure` to help with this. You use it to say that you expect this assertion to fail. When the test runs and the assertion fails, testtools will report it as an "expected failure".

Here's an example:

```
def test_expect_failure_example(self):
    self.expectFailure(
        "cats should be dogs", self.assertEqual, 'cats', 'dogs')
```

As long as 'cats' is not equal to 'dogs', the test will be reported as an expected failure.

If ever by some miracle 'cats' becomes 'dogs', then testtools will report an "unexpected success". Unlike standard unittest, testtools treats this as something that fails the test suite, like an error or a failure.

Matchers

The built-in assertion methods are very useful, they are the bread and butter of writing tests. However, soon enough you will probably want to write your own assertions. Perhaps there are domain specific things that you want to check (e.g. assert that two widgets are aligned parallel to the flux grid), or perhaps you want to check something that could almost but not quite be found in some other standard library (e.g. assert that two paths point to the same file).

When you are in such situations, you could either make a base class for your project that inherits from `testtools.TestCase` and make sure that all of your tests derive from that, *or* you could use the testtools `Matcher` system.

Using Matchers

Here's a really basic example using stock matchers found in testtools:

```
import testtools
from testtools.matchers import Equals

class TestSquare(TestCase):
    def test_square(self):
        result = square(7)
        self.assertThat(result, Equals(49))
```

The line `self.assertThat(result, Equals(49))` is equivalent to `self.assertEqual(result, 49)` and means "assert that `result` equals 49". The difference is that `assertThat` is a more general method that takes some kind of observed value (in this case, `result`) and any matcher object (here, `Equals(49)`).

The matcher object could be absolutely anything that implements the Matcher protocol. This means that you can make more complex matchers by combining existing ones:

```
def test_square_silly(self):
    result = square(7)
    self.assertThat(result, Not(Equals(50)))
```

Which is roughly equivalent to:

```
def test_square_silly(self):
    result = square(7)
    self.assertNotEqual(result, 50)
```

`assert_that` Function

In addition to `self.assertThat`, testtools also provides the `assert_that` function in `testtools.assertions` This behaves like the method version does:

```
class TestSquare(TestCase):

    def test_square():
        result = square(7)
        assert_that(result, Equals(49))

    def test_square_silly():
        result = square(7)
        assert_that(result, Not(Equals(50)))
```

Delayed Assertions

A failure in the `self.assertThat` method will immediately fail the test: No more test code will be run after the assertion failure.

The `expectThat` method behaves the same as `assertThat` with one exception: when failing the test it does so at the end of the test code rather than when the mismatch is detected. For example:

```
import subprocess

from testtools import TestCase
from testtools.matchers import Equals


class SomeProcessTests(TestCase):

    def test_process_output(self):
        process = subprocess.Popen(
          ["my-app", "/some/path"],
          stdout=subprocess.PIPE,
          stderr=subprocess.PIPE
        )

        stdout, stderrr = process.communicate()

        self.expectThat(process.returncode, Equals(0))
        self.expectThat(stdout, Equals("Expected Output"))
        self.expectThat(stderr, Equals(""))
```

In this example, should the `expectThat` call fail, the failure will be recorded in the test result, but the test will continue as normal. If all three assertions fail, the test result will have three failures recorded, and the failure details for each failed assertion will be attached to the test result.

Stock matchers

testtools comes with many matchers built in. They can all be found in and imported from the `testtools.matchers` module.

Equals

Matches if two items are equal. For example:

```
def test_equals_example(self):
    self.assertThat([42], Equals([42]))
```

Is

Matches if two items are identical. For example:

```
def test_is_example(self):
    foo = object()
    self.assertThat(foo, Is(foo))
```

IsInstance

Adapts isinstance() to use as a matcher. For example:

```
def test_isinstance_example(self):
    class MyClass:pass
    self.assertThat(MyClass(), IsInstance(MyClass))
    self.assertThat(MyClass(), IsInstance(MyClass, str))
```

The raises helper

Matches if a callable raises a particular type of exception. For example:

```
def test_raises_example(self):
    self.assertThat(lambda: 1/0, raises(ZeroDivisionError))
```

This is actually a convenience function that combines two other matchers: *Raises* and *MatchesException*.

The IsDeprecated helper

Matches if a callable produces a warning whose message matches the specified matcher. For example:

```
def old_func(x):
    warnings.warn('old_func is deprecated use new_func instead')
    return new_func(x)

def test_warning_example(self):
    self.assertThat(
      lambda: old_func(42),
      IsDeprecated(Contains('old_func is deprecated')))
```

This is just a convenience function that combines *Warnings* and *WarningMessage*.

DocTestMatches

Matches a string as if it were the output of a doctest example. Very useful for making assertions about large chunks of text. For example:

```
import doctest

def test_doctest_example(self):
    output = "Colorless green ideas"
    self.assertThat(
        output,
        DocTestMatches("Colorless ... ideas", doctest.ELLIPSIS))
```

We highly recommend using the following flags:

```
doctest.ELLIPSIS | doctest.NORMALIZE_WHITESPACE | doctest.REPORT_NDIFF
```

GreaterThan

Matches if the given thing is greater than the thing in the matcher. For example:

```
def test_greater_than_example(self):
    self.assertThat(3, GreaterThan(2))
```

LessThan

Matches if the given thing is less than the thing in the matcher. For example:

```
def test_less_than_example(self):
    self.assertThat(2, LessThan(3))
```

StartsWith, EndsWith

These matchers check to see if a string starts with or ends with a particular substring. For example:

```
def test_starts_and_ends_with_example(self):
    self.assertThat('underground', StartsWith('und'))
    self.assertThat('underground', EndsWith('und'))
```

Contains

This matcher checks to see if the given thing contains the thing in the matcher. For example:

```
def test_contains_example(self):
    self.assertThat('abc', Contains('b'))
```

MatchesException

Matches an exc_info tuple if the exception is of the correct type. For example:

```
def test_matches_exception_example(self):
    try:
        raise RuntimeError('foo')
    except RuntimeError:
        exc_info = sys.exc_info()
    self.assertThat(exc_info, MatchesException(RuntimeError))
    self.assertThat(exc_info, MatchesException(RuntimeError('bar')))
```

Most of the time, you will want to uses *The raises helper* instead.

WarningMessage

Match against various attributes (category, message and filename to name a few) of a *warning.WarningMessage*, which can be captured by *Warnings*.

NotEquals

Matches if something is not equal to something else. Note that this is subtly different to `Not(Equals(x))`. `NotEquals(x)` will match if `y != x`, `Not(Equals(x))` will match if `not y == x`.

You only need to worry about this distinction if you are testing code that relies on badly written overloaded equality operators.

KeysEqual

Matches if the keys of one dict are equal to the keys of another dict. For example:

```
def test_keys_equal(self):
    x = {'a': 1, 'b': 2}
    y = {'a': 2, 'b': 3}
    self.assertThat(x, KeysEqual(y))
```

MatchesRegex

Matches a string against a regular expression, which is a wonderful thing to be able to do, if you think about it:

```
def test_matches_regex_example(self):
    self.assertThat('foo', MatchesRegex('fo+'))
```

HasLength

Check the length of a collection. The following assertion will fail:

```
self.assertThat([1, 2, 3], HasLength(2))
```

But this one won't:

```
self.assertThat([1, 2, 3], HasLength(3))
```

File- and path-related matchers

testtools also has a number of matchers to help with asserting things about the state of the filesystem.

PathExists

Matches if a path exists:

```
self.assertThat('/', PathExists())
```

DirExists

Matches if a path exists and it refers to a directory:

```
# This will pass on most Linux systems.
self.assertThat('/home/', DirExists())
# This will not
self.assertThat('/home/jml/some-file.txt', DirExists())
```

FileExists

Matches if a path exists and it refers to a file (as opposed to a directory):

```
# This will pass on most Linux systems.
self.assertThat('/bin/true', FileExists())
# This will not.
self.assertThat('/home/', FileExists())
```

DirContains

Matches if the given directory contains the specified files and directories. Say we have a directory `foo` that has the files `a`, `b` and `c`, then:

```
self.assertThat('foo', DirContains(['a', 'b', 'c']))
```

will match, but:

```
self.assertThat('foo', DirContains(['a', 'b']))
```

will not.

The matcher sorts both the input and the list of names we get back from the filesystem.

You can use this in a more advanced way, and match the sorted directory listing against an arbitrary matcher:

```
self.assertThat('foo', DirContains(matcher=Contains('a')))
```

FileContains

Matches if the given file has the specified contents. Say there's a file called `greetings.txt` with the contents, `Hello World!`:

```
self.assertThat('greetings.txt', FileContains("Hello World!"))
```

will match.

You can also use this in a more advanced way, and match the contents of the file against an arbitrary matcher:

```
self.assertThat('greetings.txt', FileContains(matcher=Contains('!')))
```

HasPermissions

Used for asserting that a file or directory has certain permissions. Uses octal-mode permissions for both input and matching. For example:

```
self.assertThat('/tmp', HasPermissions('1777'))
self.assertThat('id_rsa', HasPermissions('0600'))
```

This is probably more useful on UNIX systems than on Windows systems.

SamePath

Matches if two paths actually refer to the same thing. The paths don't have to exist, but if they do exist, `SamePath` will resolve any symlinks.:

```
self.assertThat('somefile', SamePath('childdir/../somefile'))
```

TarballContains

Matches the contents of a tarball. In many ways, much like `DirContains`, but instead of matching on `os.listdir` matches on `TarFile.getnames`.

Combining matchers

One great thing about matchers is that you can readily combine existing matchers to get variations on their behaviour or to quickly build more complex assertions.

Below are a few of the combining matchers that come with testtools.

Always

`testtools.matchers.`**`Always`**`()`

Always match.

That is:

```
self.assertThat(x, Always())
```

Will always match and never fail, no matter what `x` is. Most useful when passed to other higher-order matchers (e.g. *`MatchesListwise`*).

Never

`testtools.matchers.`**`Never`**`()`

Never match.

That is:

```
self.assertThat(x, Never())
```

Will never match and always fail, no matter what `x` is. Included for completeness with *`Always()`*, but if you find a use for this, let us know!

Not

Negates another matcher. For example:

```
def test_not_example(self):
    self.assertThat([42], Not(Equals("potato")))
    self.assertThat([42], Not(Is([42])))
```

If you find yourself using `Not` frequently, you may wish to create a custom matcher for it. For example:

```
IsNot = lambda x: Not(Is(x))

def test_not_example_2(self):
    self.assertThat([42], IsNot([42]))
```

Annotate

Used to add custom notes to a matcher. For example:

```
def test_annotate_example(self):
    result = 43
    self.assertThat(
        result, Annotate("Not the answer to the Question!", Equals(42)))
```

Since the annotation is only ever displayed when there is a mismatch (e.g. when `result` does not equal 42), it's a good idea to phrase the note negatively, so that it describes what a mismatch actually means.

As with *Not*, you may wish to create a custom matcher that describes a common operation. For example:

```
PoliticallyEquals = lambda x: Annotate("Death to the aristos!", Equals(x))

def test_annotate_example_2(self):
    self.assertThat("orange", PoliticallyEquals("yellow"))
```

You can have assertThat perform the annotation for you as a convenience:

```
def test_annotate_example_3(self):
    self.assertThat("orange", Equals("yellow"), "Death to the aristos!")
```

AfterPreprocessing

Used to make a matcher that applies a function to the matched object before matching. This can be used to aid in creating trivial matchers as functions, for example:

```
def test_after_preprocessing_example(self):
    def PathHasFileContent(content):
        def _read(path):
            return open(path).read()
        return AfterPreprocessing(_read, Equals(content))
    self.assertThat('/tmp/foo.txt', PathHasFileContent("Hello world!"))
```

MatchesAll

Combines many matchers to make a new matcher. The new matcher will only match things that match every single one of the component matchers.

It's much easier to understand in Python than in English:

```
def test_matches_all_example(self):
    has_und_at_both_ends = MatchesAll(StartsWith("und"), EndsWith("und"))
    # This will succeed.
    self.assertThat("underground", has_und_at_both_ends)
    # This will fail.
    self.assertThat("found", has_und_at_both_ends)
```

```
    # So will this.
    self.assertThat("undead", has_und_at_both_ends)
```

At this point some people ask themselves, "why bother doing this at all? why not just have two separate assertions?". It's a good question.

The first reason is that when a `MatchesAll` gets a mismatch, the error will include information about all of the bits that mismatched. When you have two separate assertions, as below:

```
def test_two_separate_assertions(self):
     self.assertThat("foo", StartsWith("und"))
     self.assertThat("foo", EndsWith("und"))
```

Then you get absolutely no information from the second assertion if the first assertion fails. Tests are largely there to help you debug code, so having more information in error messages is a big help.

The second reason is that it is sometimes useful to give a name to a set of matchers. `has_und_at_both_ends` is a bit contrived, of course, but it is clear. The `FileExists` and `DirExists` matchers included in testtools are perhaps better real examples.

If you want only the first mismatch to be reported, pass `first_only=True` as a keyword parameter to `MatchesAll`.

MatchesAny

Like *MatchesAll*, `MatchesAny` combines many matchers to make a new matcher. The difference is that the new matchers will match a thing if it matches *any* of the component matchers.

For example:

```
def test_matches_any_example(self):
    self.assertThat(42, MatchesAny(Equals(5), Not(Equals(6))))
```

AllMatch

Matches many values against a single matcher. Can be used to make sure that many things all meet the same condition:

```
def test_all_match_example(self):
    self.assertThat([2, 3, 5, 7], AllMatch(LessThan(10)))
```

If the match fails, then all of the values that fail to match will be included in the error message.

In some ways, this is the converse of *MatchesAll*.

MatchesListwise

Where `MatchesAny` and `MatchesAll` combine many matchers to match a single value, `MatchesListwise` combines many matches to match many values.

For example:

```
def test_matches_listwise_example(self):
    self.assertThat(
        [1, 2, 3], MatchesListwise(map(Equals, [1, 2, 3])))
```

This is useful for writing custom, domain-specific matchers.

If you want only the first mismatch to be reported, pass `first_only=True` to `MatchesListwise`.

MatchesSetwise

Combines many matchers to match many values, without regard to their order.

Here's an example:

```
def test_matches_setwise_example(self):
    self.assertThat(
        [1, 2, 3], MatchesSetwise(Equals(2), Equals(3), Equals(1)))
```

Much like `MatchesListwise`, best used for writing custom, domain-specific matchers.

MatchesStructure

Creates a matcher that matches certain attributes of an object against a pre-defined set of matchers.

It's much easier to understand in Python than in English:

```
def test_matches_structure_example(self):
    foo = Foo()
    foo.a = 1
    foo.b = 2
    matcher = MatchesStructure(a=Equals(1), b=Equals(2))
    self.assertThat(foo, matcher)
```

Since all of the matchers used were `Equals`, we could also write this using the `byEquality` helper:

```
def test_matches_structure_example(self):
    foo = Foo()
    foo.a = 1
    foo.b = 2
    matcher = MatchesStructure.byEquality(a=1, b=2)
    self.assertThat(foo, matcher)
```

`MatchesStructure.fromExample` takes an object and a list of attributes and creates a `MatchesStructure` matcher where each attribute of the matched object must equal each attribute of the example object. For example:

```
matcher = MatchesStructure.fromExample(foo, 'a', 'b')
```

is exactly equivalent to `matcher` in the previous example.

MatchesPredicate

Sometimes, all you want to do is create a matcher that matches if a given function returns True, and mismatches if it returns False.

For example, you might have an `is_prime` function and want to make a matcher based on it:

```
def test_prime_numbers(self):
    IsPrime = MatchesPredicate(is_prime, '%s is not prime.')
    self.assertThat(7, IsPrime)
    self.assertThat(1983, IsPrime)
```

```
    # This will fail.
    self.assertThat(42, IsPrime)
```

Which will produce the error message:

```
Traceback (most recent call last):
  File "...", line ..., in test_prime_numbers
    self.assertThat(42, IsPrime)
MismatchError: 42 is not prime.
```

MatchesPredicateWithParams

Sometimes you can't use a trivial predicate and instead need to pass in some parameters each time. In that case, MatchesPredicateWithParams is your go-to tool for creating ad hoc matchers. MatchesPredicateWithParams takes a predicate function and message and returns a factory to produce matchers from that. The predicate needs to return a boolean (or any truthy object), and accept the object to match + whatever was passed into the factory.

For example, you might have an `divisible` function and want to make a matcher based on it:

```
def test_divisible_numbers(self):
    IsDivisibleBy = MatchesPredicateWithParams(
        divisible, '{0} is not divisible by {1}')
    self.assertThat(7, IsDivisibleBy(1))
    self.assertThat(7, IsDivisibleBy(7))
    self.assertThat(7, IsDivisibleBy(2))
    # This will fail.
```

Which will produce the error message:

```
Traceback (most recent call last):
  File "...", line ..., in test_divisible
    self.assertThat(7, IsDivisibleBy(2))
MismatchError: 7 is not divisible by 2.
```

Raises

Takes whatever the callable raises as an exc_info tuple and matches it against whatever matcher it was given. For example, if you want to assert that a callable raises an exception of a given type:

```
def test_raises_example(self):
    self.assertThat(
        lambda: 1/0, Raises(MatchesException(ZeroDivisionError)))
```

Although note that this could also be written as:

```
def test_raises_example_convenient(self):
    self.assertThat(lambda: 1/0, raises(ZeroDivisionError))
```

See also *MatchesException* and *the raises helper*

Warnings

Captures all warnings produced by a callable as a list of *warning.WarningMessage* and matches against it. For example, if you want to assert that a callable emits exactly one warning:

```
def soon_to_be_old_func():
    warnings.warn('soon_to_be_old_func will be deprecated next version',
                  PendingDeprecationWarning, 2)

def test_warnings_example(self):
    self.assertThat(
        soon_to_be_old_func,
        Warnings(HasLength(1)))
```

Deprecating something and making an assertion about the deprecation message is a very common test with which *the IsDeprecated helper* can assist. For making more specific matches against warnings *WarningMessage* can construct a matcher that can be combined with *Warnings*.

Writing your own matchers

Combining matchers is fun and can get you a very long way indeed, but sometimes you will have to write your own. Here's how.

You need to make two closely-linked objects: a `Matcher` and a `Mismatch`. The `Matcher` knows how to actually make the comparison, and the `Mismatch` knows how to describe a failure to match.

Here's an example matcher:

```
class IsDivisibleBy(object):
    """Match if a number is divisible by another number."""
    def __init__(self, divider):
        self.divider = divider
    def __str__(self):
        return 'IsDivisibleBy(%s)' % (self.divider,)
    def match(self, actual):
        remainder = actual % self.divider
        if remainder != 0:
            return IsDivisibleByMismatch(actual, self.divider, remainder)
        else:
            return None
```

The matcher has a constructor that takes parameters that describe what you actually *expect*, in this case a number that other numbers ought to be divisible by. It has a `__str__` method, the result of which is displayed on failure by `assertThat` and a `match` method that does the actual matching.

`match` takes something to match against, here `actual`, and decides whether or not it matches. If it does match, then `match` must return `None`. If it does *not* match, then `match` must return a `Mismatch` object. `assertThat` will call `match` and then fail the test if it returns a non-None value. For example:

```
def test_is_divisible_by_example(self):
    # This succeeds, since IsDivisibleBy(5).match(10) returns None.
    self.assertThat(10, IsDivisibleBy(5))
    # This fails, since IsDivisibleBy(7).match(10) returns a mismatch.
    self.assertThat(10, IsDivisibleBy(7))
```

The mismatch is responsible for what sort of error message the failing test generates. Here's an example mismatch:

```
class IsDivisibleByMismatch(object):
    def __init__(self, number, divider, remainder):
        self.number = number
        self.divider = divider
        self.remainder = remainder
```

```
    def describe(self):
        return "%r is not divisible by %r, %r remains" % (
            self.number, self.divider, self.remainder)

    def get_details(self):
        return {}
```

The mismatch takes information about the mismatch, and provides a `describe` method that assembles all of that into a nice error message for end users. You can use the `get_details` method to provide extra, arbitrary data with the mismatch (e.g. the contents of a log file). Most of the time it's fine to just return an empty dict. You can read more about *Details* elsewhere in this document.

Sometimes you don't need to create a custom mismatch class. In particular, if you don't care *when* the description is calculated, then you can just do that in the Matcher itself like this:

```
def match(self, actual):
    remainder = actual % self.divider
    if remainder != 0:
        return Mismatch(
            "%r is not divisible by %r, %r remains" % (
                actual, self.divider, remainder))
    else:
        return None
```

When writing a `describe` method or constructing a `Mismatch` object the code should ensure it only emits printable unicode. As this output must be combined with other text and forwarded for presentation, letting through non-ascii bytes of ambiguous encoding or control characters could throw an exception or mangle the display. In most cases simply avoiding the `%s` format specifier and using `%r` instead will be enough. For examples of more complex formatting see the `testtools.matchers` implementatons.

Details

As we may have mentioned once or twice already, one of the great benefits of automated tests is that they help find, isolate and debug errors in your system.

Frequently however, the information provided by a mere assertion failure is not enough. It's often useful to have other information: the contents of log files; what queries were run; benchmark timing information; what state certain subsystem components are in and so forth.

testtools calls all of these things "details" and provides a single, powerful mechanism for including this information in your test run.

Here's an example of how to add them:

```
from testtools import TestCase
from testtools.content import text_content

class TestSomething(TestCase):

    def test_thingy(self):
        self.addDetail('arbitrary-color-name', text_content("blue"))
        1 / 0 # Gratuitous error!
```

A detail an arbitrary piece of content given a name that's unique within the test. Here the name is `arbitrary-color-name` and the content is `text_content("blue")`. The name can be any text string,

and the content can be any `testtools.content.Content` object.

When the test runs, testtools will show you something like this:

```
======================================================================
ERROR: exampletest.TestSomething.test_thingy
----------------------------------------------------------------------
arbitrary-color-name: {{{blue}}}

Traceback (most recent call last):
  File "exampletest.py", line 8, in test_thingy
    1 / 0 # Gratuitous error!
ZeroDivisionError: integer division or modulo by zero
------------
Ran 1 test in 0.030s
```

As you can see, the detail is included as an attachment, here saying that our arbitrary-color-name is "blue".

Content

For the actual content of details, testtools uses its own MIME-based Content object. This allows you to attach any information that you could possibly conceive of to a test, and allows testtools to use or serialize that information.

The basic `testtools.content.Content` object is constructed from a `testtools.content.ContentType` and a nullary callable that must return an iterator of chunks of bytes that the content is made from.

So, to make a Content object that is just a simple string of text, you can do:

```
from testtools.content import Content
from testtools.content_type import ContentType

text = Content(ContentType('text', 'plain'), lambda: ["some text"])
```

Because adding small bits of text content is very common, there's also a convenience method:

```
text = text_content("some text")
```

To make content out of an image stored on disk, you could do something like:

```
image = Content(ContentType('image', 'png'), lambda: open('foo.png').read())
```

Or you could use the convenience function:

```
image = content_from_file('foo.png', ContentType('image', 'png'))
```

The `lambda` helps make sure that the file is opened and the actual bytes read only when they are needed – by default, when the test is finished. This means that tests can construct and add Content objects freely without worrying too much about how they affect run time.

A realistic example

A very common use of details is to add a log file to failing tests. Say your project has a server represented by a class `SomeServer` that you can start up and shut down in tests, but runs in another process. You want to test interaction with that server, and whenever the interaction fails, you want to see the client-side error *and* the logs from the server-side. Here's how you might do it:

```
from testtools import TestCase
from testtools.content import attach_file, Content
from testtools.content_type import UTF8_TEXT

from myproject import SomeServer

class SomeTestCase(TestCase):

    def setUp(self):
        super(SomeTestCase, self).setUp()
        self.server = SomeServer()
        self.server.start_up()
        self.addCleanup(self.server.shut_down)
        self.addCleanup(attach_file, self.server.logfile, self)

    def attach_log_file(self):
        self.addDetail(
            'log-file',
            Content(UTF8_TEXT,
                    lambda: open(self.server.logfile, 'r').readlines()))

    def test_a_thing(self):
        self.assertEqual("cool", self.server.temperature)
```

This test will attach the log file of `SomeServer` to each test that is run. testtools will only display the log file for failing tests, so it's not such a big deal.

If the act of adding at detail is expensive, you might want to use *addOnException* so that you only do it when a test actually raises an exception.

Controlling test execution

addCleanup

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

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

This is particularly useful if you have some sort of factory in your test:

```
def make_locked_foo(self):
    foo = Foo()
    foo.lock()
    self.addCleanup(foo.unlock)
    return foo

def test_frotz_a_foo(self):
    foo = self.make_locked_foo()
    foo.frotz()
    self.assertEqual(foo.frotz_count, 1)
```

Any extra arguments or keyword arguments passed to `addCleanup` are passed to the callable at cleanup time.

Cleanups can also report multiple errors, if appropriate by wrapping them in a `testtools.MultipleExceptions` object:

```
raise MultipleExceptions(exc_info1, exc_info2)
```

Fixtures

Tests often depend on a system being set up in a certain way, or having certain resources available to them. Perhaps a test needs a connection to the database or access to a running external server.

One common way of doing this is to do:

```
class SomeTest(TestCase):
    def setUp(self):
        super(SomeTest, self).setUp()
        self.server = Server()
        self.server.setUp()
        self.addCleanup(self.server.tearDown)
```

testtools provides a more convenient, declarative way to do the same thing:

```
class SomeTest(TestCase):
    def setUp(self):
        super(SomeTest, self).setUp()
        self.server = self.useFixture(Server())
```

`useFixture(fixture)` calls `setUp` on the fixture, schedules a clean up to clean it up, and schedules a clean up to attach all *details* held by the fixture to the test case. The fixture object must meet the `fixtures.Fixture` protocol (version 0.3.4 or newer, see fixtures).

If you have anything beyond the most simple test set up, we recommend that you put this set up into a `Fixture` class. Once there, the fixture can be easily re-used by other tests and can be combined with other fixtures to make more complex resources.

Skipping tests

Many reasons exist to skip a test: a dependency might be missing; a test might be too expensive and thus should not berun while on battery power; or perhaps the test is testing an incomplete feature.

`TestCase.skipTest` is a simple way to have a test stop running and be reported as a skipped test, rather than a success, error or failure. For example:

```
def test_make_symlink(self):
    symlink = getattr(os, 'symlink', None)
    if symlink is None:
        self.skipTest("No symlink support")
    symlink(whatever, something_else)
```

Using `skipTest` means that you can make decisions about what tests to run as late as possible, and close to the actual tests. Without it, you might be forced to use convoluted logic during test loading, which is a bit of a mess.

Legacy skip support

If you are using this feature when running your test suite with a legacy `TestResult` object that is missing the `addSkip` method, then the `addError` method will be invoked instead. If you are using a test result from testtools, you do not have to worry about this.

In older versions of testtools, `skipTest` was known as `skip`. Since Python 2.7 added `skipTest` support, the `skip` name is now deprecated.

addOnException

Sometimes, you might wish to do something only when a test fails. Perhaps you need to run expensive diagnostic routines or some such. `TestCase.addOnException` allows you to easily do just this. For example:

```
class SomeTest(TestCase):
    def setUp(self):
        super(SomeTest, self).setUp()
        self.server = self.useFixture(SomeServer())
        self.addOnException(self.attach_server_diagnostics)

    def attach_server_diagnostics(self, exc_info):
        self.server.prep_for_diagnostics() # Expensive!
        self.addDetail('server-diagnostics', self.server.get_diagnostics)

    def test_a_thing(self):
        self.assertEqual('cheese', 'chalk')
```

In this example, `attach_server_diagnostics` will only be called when a test fails. It is given the exc_info tuple of the error raised by the test, just in case it is needed.

Twisted support

testtools provides support for running Twisted tests – tests that return a Deferred and rely on the Twisted reactor. See *Twisted support*.

force_failure

Setting the `testtools.TestCase.force_failure` instance variable to `True` will cause the test to be marked as a failure, but won't stop the test code from running (see *Delayed Test Failure*).

Test helpers

testtools comes with a few little things that make it a little bit easier to write tests.

TestCase.patch

`patch` is a convenient way to monkey-patch a Python object for the duration of your test. It's especially useful for testing legacy code. e.g.:

```
def test_foo(self):
    my_stream = StringIO()
    self.patch(sys, 'stderr', my_stream)
    run_some_code_that_prints_to_stderr()
    self.assertEqual('', my_stream.getvalue())
```

The call to `patch` above masks `sys.stderr` with `my_stream` so that anything printed to stderr will be captured in a StringIO variable that can be actually tested. Once the test is done, the real `sys.stderr` is restored to its rightful place.

Creation methods

Often when writing unit tests, you want to create an object that is a completely normal instance of its type. You don't want there to be anything special about its properties, because you are testing generic behaviour rather than specific conditions.

A lot of the time, test authors do this by making up silly strings and numbers and passing them to constructors (e.g. 42, 'foo', "bar" etc), and that's fine. However, sometimes it's useful to be able to create arbitrary objects at will, without having to make up silly sample data.

To help with this, `testtools.TestCase` implements creation methods called `getUniqueString` and `getUniqueInteger`. They return strings and integers that are unique within the context of the test that can be used to assemble more complex objects. Here's a basic example where `getUniqueString` is used instead of saying "foo" or "bar" or whatever:

```
class SomeTest(TestCase):

    def test_full_name(self):
        first_name = self.getUniqueString()
        last_name = self.getUniqueString()
        p = Person(first_name, last_name)
        self.assertEqual(p.full_name, "%s %s" % (first_name, last_name))
```

And here's how it could be used to make a complicated test:

```
class TestCoupleLogic(TestCase):

    def make_arbitrary_person(self):
        return Person(self.getUniqueString(), self.getUniqueString())

    def test_get_invitation(self):
        a = self.make_arbitrary_person()
        b = self.make_arbitrary_person()
        couple = Couple(a, b)
        event_name = self.getUniqueString()
        invitation = couple.get_invitation(event_name)
        self.assertEqual(
            invitation,
            "We invite %s and %s to %s" % (
                a.full_name, b.full_name, event_name))
```

Essentially, creation methods like these are a way of reducing the number of assumptions in your tests and communicating to test readers that the exact details of certain variables don't actually matter.

See pages 419-423 of xUnit Test Patterns by Gerard Meszaros for a detailed discussion of creation methods.

`testcase.generate_unique_text()` is similar to `getUniqueString`, except it generates text that contains unicode characters. The value will be a `unicode` object in Python 2 and a `str` object in Python 3.

Test attributes

Inspired by the `nosetests attr` plugin, testtools provides support for marking up test methods with attributes, which are then exposed in the test id and can be used when filtering tests by id. (e.g. via `--load-list`):

```
from testtools.testcase import attr, WithAttributes

class AnnotatedTests(WithAttributes, TestCase):

    @attr('simple')
    def test_one(self):
        pass

    @attr('more', 'than', 'one')
    def test_two(self):
        pass

    @attr('or')
    @attr('stacked')
    def test_three(self):
        pass
```

General helpers

Conditional imports

Lots of the time we would like to conditionally import modules. testtools uses the small library extras to do this. This used to be part of testtools.

Instead of:

```
try:
    from twisted.internet import defer
except ImportError:
    defer = None
```

You can do:

```
defer = try_import('twisted.internet.defer')
```

Instead of:

```
try:
    from StringIO import StringIO
except ImportError:
    from io import StringIO
```

You can do:

```
StringIO = try_imports(['StringIO.StringIO', 'io.StringIO'])
```

Safe attribute testing

`hasattr` is broken on many versions of Python. The helper `safe_hasattr` can be used to safely test whether an object has a particular attribute. Like `try_import` this used to be in testtools but is now in extras.

Nullary callables

Sometimes you want to be able to pass around a function with the arguments already specified. The normal way of doing this in Python is:

```
nullary = lambda: f(*args, **kwargs)
nullary()
```

Which is mostly good enough, but loses a bit of debugging information. If you take the `repr()` of `nullary`, you're only told that it's a lambda, and you get none of the juicy meaning that you'd get from the `repr()` of `f`.

The solution is to use `Nullary` instead:

```
nullary = Nullary(f, *args, **kwargs)
nullary()
```

Here, `repr(nullary)` will be the same as `repr(f)`.

CHAPTER 3

testtools for framework folk

Introduction

In addition to having many features *for test authors*, testtools also has many bits and pieces that are useful for folk who write testing frameworks.

If you are the author of a test runner, are working on a very large unit-tested project, are trying to get one testing framework to play nicely with another or are hacking away at getting your test suite to run in parallel over a heterogenous cluster of machines, this guide is for you.

This manual is a summary. You can get details by consulting the *testtools API docs*.

Extensions to TestCase

In addition to the `TestCase` specific methods, we have extensions for `TestSuite` that also apply to `TestCase` (because `TestCase` and `TestSuite` follow the Composite pattern).

Custom exception handling

testtools provides a way to control how test exceptions are handled. To do this, add a new exception to `self.exception_handlers` on a `testtools.TestCase`. For example:

```
>>> self.exception_handlers.insert(-1, (ExceptionClass, handler)).
```

Having done this, if any of `setUp`, `tearDown`, or the test method raise `ExceptionClass`, `handler` will be called with the test case, test result and the raised exception.

Use this if you want to add a new kind of test result, that is, if you think that `addError`, `addFailure` and so forth are not enough for your needs.

Controlling test execution

If you want to control more than just how exceptions are raised, you can provide a custom `RunTest` to a `TestCase`. The `RunTest` object can change everything about how the test executes.

To work with `testtools.TestCase`, a `RunTest` must have a factory that takes a test and an optional list of exception handlers and an optional last_resort handler. Instances returned by the factory must have a `run()` method that takes an optional `TestResult` object.

The default is `testtools.runtest.RunTest`, which calls `setUp`, the test method, `tearDown` and clean ups (see *addCleanup*) in the normal, vanilla way that Python's standard unittest does.

To specify a `RunTest` for all the tests in a `TestCase` class, do something like this:

```
class SomeTests(TestCase):
    run_tests_with = CustomRunTestFactory
```

To specify a `RunTest` for a specific test in a `TestCase` class, do:

```
class SomeTests(TestCase):
    @run_test_with(CustomRunTestFactory, extra_arg=42, foo='whatever')
    def test_something(self):
        pass
```

In addition, either of these can be overridden by passing a factory in to the `TestCase` constructor with the optional `runTest` argument.

Test renaming

`testtools.clone_test_with_new_id` is a function to copy a test case instance to one with a new name. This is helpful for implementing test parameterization.

Delayed Test Failure

Setting the `testtools.TestCase.force_failure` instance variable to True will cause `testtools.RunTest` to fail the test case after the test has finished. This is useful when you want to cause a test to fail, but don't want to prevent the remainder of the test code from being executed.

Exception formatting

Testtools `TestCase` instances format their own exceptions. The attribute `__testtools_tb_locals__` controls whether to include local variables in the formatted exceptions.

Test placeholders

Sometimes, it's useful to be able to add things to a test suite that are not actually tests. For example, you might wish to represents import failures that occur during test discovery as tests, so that your test result object doesn't have to do special work to handle them nicely.

testtools provides two such objects, called "placeholders": `PlaceHolder` and `ErrorHolder`. `PlaceHolder` takes a test id and an optional description. When it's run, it succeeds. `ErrorHolder` takes a test id, and error and an optional short description. When it's run, it reports that error.

These placeholders are best used to log events that occur outside the test suite proper, but are still very relevant to its results.

e.g.:

```
>>> suite = TestSuite()
>>> suite.add(PlaceHolder('I record an event'))
>>> suite.run(TextTestResult(verbose=True))
I record an event                                                    [OK]
```

Test instance decorators

DecorateTestCaseResult

This object calls out to your code when `run` / `__call__` are called and allows the result object that will be used to run the test to be altered. This is very useful when working with a test runner that doesn't know your test case requirements. For instance, it can be used to inject a `unittest2` compatible adapter when someone attempts to run your test suite with a `TestResult` that does not support `addSkip` or other `unittest2` methods. Similarly it can aid the migration to `StreamResult`.

e.g.:

```
>>> suite = TestSuite()
>>> suite = DecorateTestCaseResult(suite, ExtendedToOriginalDecorator)
```

Extensions to TestResult

StreamResult

`StreamResult` is a new API for dealing with test case progress that supports concurrent and distributed testing without the various issues that `TestResult` has such as buffering in multiplexers.

The design has several key principles:

- Nothing that requires up-front knowledge of all tests.
- Deal with tests running in concurrent environments, potentially distributed across multiple processes (or even machines). This implies allowing multiple tests to be active at once, supplying time explicitly, being able to differentiate between tests running in different contexts and removing any assumption that tests are necessarily in the same process.
- Make the API as simple as possible - each aspect should do one thing well.

The `TestResult` API this is intended to replace has three different clients.

- Each executing `TestCase` notifies the `TestResult` about activity.
- The testrunner running tests uses the API to find out whether the test run had errors, how many tests ran and so on.
- Finally, each `TestCase` queries the `TestResult` to see whether the test run should be aborted.

With `StreamResult` we need to be able to provide a `TestResult` compatible adapter (`StreamToExtendedDecorator`) to allow incremental migration. However, we don't need to conflate things long term. So - we define three separate APIs, and merely mix them together to provide the

`StreamToExtendedDecorator`. `StreamResult` is the first of these APIs - meeting the needs of `TestCase` clients. It handles events generated by running tests. See the API documentation for `testtools.StreamResult` for details.

StreamSummary

Secondly we define the `StreamSummary` API which takes responsibility for collating errors, detecting incomplete tests and counting tests. This provides a compatible API with those aspects of `TestResult`. Again, see the API documentation for `testtools.StreamSummary`.

TestControl

Lastly we define the `TestControl` API which is used to provide the `shouldStop` and `stop` elements from `TestResult`. Again, see the API documentation for `testtools.TestControl`. `TestControl` can be paired with a `StreamFailFast` to trigger aborting a test run when a failure is observed. Aborting multiple workers in a distributed environment requires hooking whatever signalling mechanism the distributed environment has up to a `TestControl` in each worker process.

StreamTagger

A `StreamResult` filter that adds or removes tags from events:

```
>>> from testtools import StreamTagger
>>> sink = StreamResult()
>>> result = StreamTagger([sink], set(['add']), set(['discard']))
>>> result.startTestRun()
>>> # Run tests against result here.
>>> result.stopTestRun()
```

StreamToDict

A simplified API for dealing with `StreamResult` streams. Each test is buffered until it completes and then reported as a trivial dict. This makes writing analysers very easy - you can ignore all the plumbing and just work with the result. e.g.:

```
>>> from testtools import StreamToDict
>>> def handle_test(test_dict):
...     print(test_dict['id'])
>>> result = StreamToDict(handle_test)
>>> result.startTestRun()
>>> # Run tests against result here.
>>> # At stopTestRun() any incomplete buffered tests are announced.
>>> result.stopTestRun()
```

ExtendedToStreamDecorator

This is a hybrid object that combines both the `Extended` and `Stream` `TestResult` APIs into one class, but only emits `StreamResult` events. This is useful when a `StreamResult` stream is desired, but you cannot be sure that the tests which will run have been updated to the `StreamResult` API.

StreamToExtendedDecorator

This is a simple converter that emits the `ExtendedTestResult` API in response to events from the `StreamResult` API. Useful when outputting `StreamResult` events from a `TestCase` but the supplied `TestResult` does not support the `status` and `file` methods.

StreamToQueue

This is a `StreamResult` decorator for reporting tests from multiple threads at once. Each method submits an event to a supplied Queue object as a simple dict. See `ConcurrentStreamTestSuite` for a convenient way to use this.

TimestampingStreamResult

This is a `StreamResult` decorator for adding timestamps to events that lack them. This allows writing the simplest possible generators of events and passing the events via this decorator to get timestamped data. As long as no buffering/queueing or blocking happen before the timestamper sees the event the timestamp will be as accurate as if the original event had it.

StreamResultRouter

This is a `StreamResult` which forwards events to an arbitrary set of target `StreamResult` objects. Events that have no forwarding rule are passed onto an fallback `StreamResult` for processing. The mapping can be changed at runtime, allowing great flexibility and responsiveness to changes. Because The mapping can change dynamically and there could be the same recipient for two different maps, `startTestRun` and `stopTestRun` handling is fine grained and up to the user.

If no fallback has been supplied, an unroutable event will raise an exception.

For instance:

```
>>> router = StreamResultRouter()
>>> sink = doubles.StreamResult()
>>> router.add_rule(sink, 'route_code_prefix', route_prefix='0',
...     consume_route=True)
>>> router.status(test_id='foo', route_code='0/1', test_status='uxsuccess')
```

Would remove the `0/` from the route_code and forward the event like so:

```
>>> sink.status('test_id=foo', route_code='1', test_status='uxsuccess')
```

See `pydoc testtools.StreamResultRouter` for details.

TestResult.addSkip

This method is called on result objects when a test skips. The `testtools.TestResult` class records skips in its `skip_reasons` instance dict. The can be reported on in much the same way as succesful tests.

TestResult.time

This method controls the time used by a `TestResult`, permitting accurate timing of test results gathered on different machines or in different threads. See pydoc testtools.TestResult.time for more details.

ThreadsafeForwardingResult

A `TestResult` which forwards activity to another test result, but synchronises on a semaphore to ensure that all the activity for a single test arrives in a batch. This allows simple TestResults which do not expect concurrent test reporting to be fed the activity from multiple test threads, or processes.

Note that when you provide multiple errors for a single test, the target sees each error as a distinct complete test.

MultiTestResult

A test result that dispatches its events to many test results. Use this to combine multiple different test result objects into one test result object that can be passed to `TestCase.run()` or similar. For example:

```
a = TestResult()
b = TestResult()
combined = MultiTestResult(a, b)
combined.startTestRun()  # Calls a.startTestRun() and b.startTestRun()
```

Each of the methods on `MultiTestResult` will return a tuple of whatever the component test results return.

TestResultDecorator

Not strictly a `TestResult`, but something that implements the extended `TestResult` interface of testtools. It can be subclassed to create objects that wrap `TestResults`.

TextTestResult

A `TestResult` that provides a text UI very similar to the Python standard library UI. Key differences are that its supports the extended outcomes and details API, and is completely encapsulated into the result object, permitting it to be used without a 'TestRunner' object. Not all the Python 2.7 outcomes are displayed (yet). It is also a 'quiet' result with no dots or verbose mode. These limitations will be corrected soon.

ExtendedToOriginalDecorator

Adapts legacy `TestResult` objects, such as those found in older Pythons, to meet the testtools `TestResult` API.

Test Doubles

In testtools.testresult.doubles there are three test doubles that testtools uses for its own testing: `Python26TestResult`, `Python27TestResult`, `ExtendedTestResult`. These TestResult objects implement a single variation of the TestResult API each, and log activity to a list `self._events`. These are made available for the convenience of people writing their own extensions.

startTestRun and stopTestRun

Python 2.7 added hooks `startTestRun` and `stopTestRun` which are called before and after the entire test run. 'stopTestRun' is particularly useful for test results that wish to produce summary output.

`testtools.TestResult` provides default `startTestRun` and `stopTestRun` methods, and he default testtools runner will call these methods appropriately.

The `startTestRun` method will reset any errors, failures and so forth on the result, making the result object look as if no tests have been run.

Extensions to TestSuite

ConcurrentTestSuite

A TestSuite for parallel testing. This is used in conjuction with a helper that runs a single suite in some parallel fashion (for instance, forking, handing off to a subprocess, to a compute cloud, or simple threads). ConcurrentTestSuite uses the helper to get a number of separate runnable objects with a run(result), runs them all in threads using the ThreadsafeForwardingResult to coalesce their activity.

ConcurrentStreamTestSuite

A variant of ConcurrentTestSuite that uses the new StreamResult API instead of the TestResult API. ConcurrentStreamTestSuite coordinates running some number of test/suites concurrently, with one StreamToQueue per test/suite.

Each test/suite gets given its own ExtendedToStreamDecorator + TimestampingStreamResult wrapped StreamToQueue instance, forwarding onto the StreamResult that ConcurrentStreamTestSuite.run was called with.

ConcurrentStreamTestSuite is a thin shim and it is easy to implement your own specialised form if that is needed.

FixtureSuite

A test suite that sets up a fixture before running any tests, and then tears it down after all of the tests are run. The fixture is *not* made available to any of the tests due to there being no standard channel for suites to pass information to the tests they contain (and we don't have enough data on what such a channel would need to achieve to design a good one yet - or even decide if it is a good idea).

sorted_tests

Given the composite structure of TestSuite / TestCase, sorting tests is problematic - you can't tell what functionality is embedded into custom Suite implementations. In order to deliver consistent test orders when using test discovery (see http://bugs.python.org/issue16709), testtools flattens and sorts tests that have the standard TestSuite, and defines a new method sort_tests, which can be used by non-standard TestSuites to know when they should sort their tests. An example implementation can be seen at `FixtureSuite.sorted_tests`.

If there are duplicate test ids in a suite, ValueError will be raised.

filter_by_ids

Similarly to `sorted_tests` running a subset of tests is problematic - the standard run interface provides no way to limit what runs. Rather than confounding the two problems (selection and execution) we defined a method that filters the tests in a suite (or a case) by their unique test id. If you a writing custom wrapping suites, consider implementing filter_by_ids to support this (though most wrappers that subclass `unittest.TestSuite` will work just fine [see `testtools.testsuite.filter_by_ids` for details.]

Extensions to TestRunner

To facilitate custom listing of tests, `testtools.run.TestProgram` attempts to call `list` on the `TestRunner`, falling back to a generic implementation if it is not present.

CHAPTER 4

Twisted support

testtools provides support for testing Twisted code.

Matching Deferreds

testtools provides support for making assertions about synchronous `Deferred`s.

A "synchronous" `Deferred` is one that does not need the reactor or any other asynchronous process in order to fire.

Normal application code can't know when a `Deferred` is going to fire, because that is generally left up to the reactor. Well-written unit tests provide fake reactors, or don't use the reactor at all, so that `Deferred`s fire synchronously.

These matchers allow you to make assertions about when and how `Deferred`s fire, and about what values they fire with.

See also Testing Deferreds without the reactor and the Deferred howto.

`testtools.twistedsupport.`**`succeeded`**(*matcher*)
Match a Deferred that has fired successfully.

For example:

```
fires_with_the_answer = succeeded(Equals(42))
deferred = defer.succeed(42)
assert_that(deferred, fires_with_the_answer)
```

This assertion will pass. However, if `deferred` had fired with a different value, or had failed, or had not fired at all, then it would fail.

Use this instead of `twisted.trial.unittest.SynchronousTestCase.successResultOf()`.

Parameters **`matcher`** – A matcher to match against the result of a `Deferred`.

Returns A matcher that can be applied to a synchronous `Deferred`.

`testtools.twistedsupport.`**`failed`**(*matcher*)
Match a Deferred that has failed.

For example:

```
error = RuntimeError('foo')
fails_at_runtime = failed(
    AfterPreprocessing(lambda f: f.value, Equals(error)))
deferred = defer.fail(error)
assert_that(deferred, fails_at_runtime)
```

This assertion will pass. However, if `deferred` had fired successfully, had failed with a different error, or had not fired at all, then it would fail.

Use this instead of `twisted.trial.unittest.SynchronousTestCase.failureResultOf()`.

Parameters **matcher** – A matcher to match against the result of a failing `Deferred`.

Returns A matcher that can be applied to a synchronous `Deferred`.

`testtools.twistedsupport.`**has_no_result**`()`

Match a Deferred that has not yet fired.

For example, this will pass:

```
assert_that(defer.Deferred(), has_no_result())
```

But this will fail:

```
>>> assert_that(defer.succeed(None), has_no_result())
Traceback (most recent call last):
  ...
  File "testtools/assertions.py", line 22, in assert_that
    raise MismatchError(matchee, matcher, mismatch, verbose)
testtools.matchers._impl.MismatchError: No result expected on <Deferred at ...␣
↪current result: None>, found None instead
```

As will this:

```
>>> assert_that(defer.fail(RuntimeError('foo')), has_no_result())
Traceback (most recent call last):
  ...
  File "testtools/assertions.py", line 22, in assert_that
    raise MismatchError(matchee, matcher, mismatch, verbose)
testtools.matchers._impl.MismatchError: No result expected on <Deferred at ...␣
↪current result: <twisted.python.failure.Failure <type 'exceptions.RuntimeError'>
↪>>, found <twisted.python.failure.Failure <type 'exceptions.RuntimeError'>>␣
↪instead
```

Running tests in the reactor

testtools provides support for running asynchronous Twisted tests: tests that return a `Deferred` and run the reactor until it fires and its callback chain is completed.

Here's how to use it:

```
from testtools import TestCase
from testtools.twistedsupport import AsynchronousDeferredRunTest

class MyTwistedTests(TestCase):
```

```
    run_tests_with = AsynchronousDeferredRunTest

    def test_foo(self):
        # ...
        return d
```

Note that you do *not* have to use a special base `TestCase` in order to run Twisted tests, you should just use the regular *testtools.TestCase* base class.

You can also run individual tests within a test case class using the Twisted test runner:

```
class MyTestsSomeOfWhichAreTwisted(TestCase):

    def test_normal(self):
        pass

    @run_test_with(AsynchronousDeferredRunTest)
    def test_twisted(self):
        # ...
        return d
```

See *AsynchronousDeferredRunTest* and *AsynchronousDeferredRunTestForBrokenTwisted* for more information.

Controlling the Twisted logs

Users of Twisted Trial will be accustomed to all tests logging to `_trial_temp/test.log`. By default, *AsynchronousDeferredRunTest* will *not* do this, but will instead:

1. suppress all messages logged during the test run
2. attach them as the `twisted-log` detail (see *Details*) which is shown if the test fails

The first behavior is controlled by the `suppress_twisted_logging` parameter to *AsynchronousDeferredRunTest*, which is set to `True` by default. The second is controlled by the `store_twisted_logs` parameter, which is also `True` by default.

If `store_twisted_logs` is set to `False`, you can still get the logs attached as a detail by using the *CaptureTwistedLogs* fixture. Using the *CaptureTwistedLogs* fixture is equivalent to setting `store_twisted_logs` to `True`.

For example:

```
class DoNotCaptureLogsTests(TestCase):
    run_tests_with = partial(AsynchronousDeferredRunTest,
                             store_twisted_logs=False)

    def test_foo(self):
        log.msg('logs from this test are not attached')

    def test_bar(self):
        self.useFixture(CaptureTwistedLogs())
        log.msg('logs from this test *are* attached')
```

Converting Trial tests to testtools tests

- Use the *AsynchronousDeferredRunTest* runner
- Make sure to upcall to `TestCase.setUp()` and `TestCase.tearDown()`
- Don't use `setUpClass` or `tearDownClass`
- Don't expect setting `.todo`, `.timeout` or `.skip` attributes to do anything
- Replace `twisted.trial.unittest.SynchronousTestCase.flushLoggedErrors()` with *flush_logged_errors()*
- Replace `twisted.trial.unittest.TestCase.assertFailure()` with *assert_fails_with()*
- Trial spins the reactor a couple of times before cleaning it up, *AsynchronousDeferredRunTest* does not. If you rely on this behavior, use *AsynchronousDeferredRunTestForBrokenTwisted*.

CHAPTER 5

Contributing to testtools

Bugs and patches

File bugs <https://bugs.launchpad.net/testtools/+filebug> on Launchpad, and *send patches <https://github.com/testing-cabal/testtools/>* on Github.

Coding style

In general, follow PEP 8 except where consistency with the standard library's unittest module would suggest otherwise.

testtools currently supports Python 2.6 and later, including Python 3.

Copyright assignment

Part of testtools raison d'etre is to provide Python with improvements to the testing code it ships. For that reason we require all contributions (that are non-trivial) to meet one of the following rules:

- be inapplicable for inclusion in Python.
- be able to be included in Python without further contact with the contributor.
- be copyright assigned to Jonathan M. Lange.

Please pick one of these and specify it when contributing code to testtools.

Licensing

All code that is not copyright assigned to Jonathan M. Lange (see Copyright Assignment above) needs to be licensed under the MIT license that testtools uses, so that testtools can ship it.

Building

Building and installing testtools requires a reasonably recent version of pip. At the time of writing, pip version 7.1.0 (which is bundled with virtualenv 13.1.0) is a good choice. To install testtools from source and all its test dependencies, install the `test` extra:

```
pip install -e .[test]
```

Installing via `python setup.py install` may not work, due to issues with easy_install.

Testing

Please write tests for every feature. This project ought to be a model example of well-tested Python code!

Take particular care to make sure the *intent* of each test is clear.

You can run tests with `make check`.

By default, testtools hides many levels of its own stack when running tests. This is for the convenience of users, who do not care about how, say, assert methods are implemented. However, when writing tests for testtools itself, it is often useful to see all levels of the stack. To do this, add `run_tests_with = FullStackRunTest` to the top of a test's class definition.

Discussion

When submitting a patch, it will help the review process a lot if there's a clear explanation of what the change does and why you think the change is a good idea. For crasher bugs, this is generally a no-brainer, but for UI bugs & API tweaks, the reason something is an improvement might not be obvious, so it's worth spelling out.

If you are thinking of implementing a new feature, you might want to have that discussion on the [mailing list](testtools-dev@lists.launchpad.net) before the patch goes up for review. This is not at all mandatory, but getting feedback early can help avoid dead ends.

Documentation

Documents are written using the Sphinx variant of reStructuredText. All public methods, functions, classes and modules must have API documentation. When changing code, be sure to check the API documentation to see if it could be improved. Before submitting changes to trunk, look over them and see if the manuals ought to be updated.

Source layout

The top-level directory contains the `testtools/` package directory, and miscellaneous files like `README.rst` and `setup.py`.

The `testtools/` directory is the Python package itself. It is separated into submodules for internal clarity, but all public APIs should be "promoted" into the top-level package by importing them in `testtools/__init__.py`. Users of testtools should never import a submodule in order to use a stable API. Unstable APIs like `testtools.matchers` and `testtools.deferredruntest` should be exported as submodules.

Tests belong in `testtools/tests/`.

Committing to trunk

Testtools is maintained using git, with its master repo at https://github.com/testing-cabal/testtools. This gives every contributor the ability to commit their work to their own branches. However permission must be granted to allow contributors to commit to the trunk branch.

Commit access to trunk is obtained by joining the testing-cabal, either as an Owner or a Committer. Commit access is contingent on obeying the testtools contribution policy, see *Copyright Assignment* above.

Code Review

All code must be reviewed before landing on trunk. The process is to create a branch on Github, and make a pull request into trunk. It will then be reviewed before it can be merged to trunk. It will be reviewed by someone:

- not the author
- a committer

As a special exception, since there are few testtools committers and thus reviews are prone to blocking, a pull request from a committer that has not been reviewed after 24 hours may be merged by that committer. When the team is larger this policy will be revisited.

Code reviewers should look for the quality of what is being submitted, including conformance with this HACKING file.

Changes which all users should be made aware of should be documented in NEWS.

We are now in full backwards compatibility mode - no more releases < 1.0.0, and breaking compatibility will require consensus on the testtools-dev mailing list. Exactly what constitutes a backwards incompatible change is vague, but coarsely:

- adding required arguments or required calls to something that used to work
- removing keyword or position arguments, removing methods, functions or modules
- changing behaviour someone may have reasonably depended on

Some things are not compatibility issues:

- changes to _ prefixed methods, functions, modules, packages.

NEWS management

The file NEWS is structured as a sorted list of releases. Each release can have a free form description and more or more sections with bullet point items. Sections in use today are 'Improvements' and 'Changes'. To ease merging between branches, the bullet points are kept alphabetically sorted. The release NEXT is permanently present at the top of the list.

Releasing

Prerequisites

Membership in the testing-cabal org on github as committer.

Membership in the pypi testtools project as maintainer.

Membership in the https://launchpad.net/~testtools-committers.

No in-progress Critical bugs on the next milestone.

Tasks

1. Choose a version number, say X.Y.Z
2. Under NEXT in NEWS add a heading with the version number X.Y.Z.
3. Possibly write a blurb into NEWS.
4. Commit the changes.
5. Tag the release, `git tag -s X.Y.Z -m "Releasing X.Y.Z"`
6. Run 'make release', this: #. Creates a source distribution and uploads to PyPI #. Ensures all Fix Committed bugs are in the release milestone #. Makes a release on Launchpad and uploads the tarball #. Marks all the Fix Committed bugs as Fix Released #. Creates a new milestone
7. If a new series has been created (e.g. 0.10.0), make the series on Launchpad.
8. Push trunk to Github, `git push --tags origin master`

CHAPTER 6

testtools API documentation

Generated reference documentation for all the public functionality of testtools.

Please *send patches* if you notice anything confusing or wrong, or that could be improved.

testtools

Extensions to the standard Python unittest library.

testtools.**clone_test_with_new_id**(*test*, *new_id*)
Copy a *TestCase*, and give the copied test a new id.

This is only expected to be used on tests that have been constructed but not executed.

class testtools.**CopyStreamResult**(*targets*)
Copies all event it receives to multiple results.

This provides an easy facility for combining multiple StreamResults.

For TestResult the equivalent class was `MultiTestResult`.

class testtools.**ConcurrentTestSuite**(*suite*, *make_tests*, *wrap_result=None*)
A TestSuite whose run() calls out to a concurrency strategy.

run(*result*)
Run the tests concurrently.

This calls out to the provided make_tests helper, and then serialises the results so that result only sees activity from one TestCase at a time.

ConcurrentTestSuite provides no special mechanism to stop the tests returned by make_tests, it is up to the make_tests to honour the shouldStop attribute on the result object they are run with, which will be set if an exception is raised in the thread which ConcurrentTestSuite.run is called in.

class testtools.**ConcurrentStreamTestSuite**(*make_tests*)
A TestSuite whose run() parallelises.

run(*result*)
Run the tests concurrently.

This calls out to the provided make_tests helper to determine the concurrency to use and to assign routing codes to each worker.

ConcurrentTestSuite provides no special mechanism to stop the tests returned by make_tests, it is up to the made tests to honour the shouldStop attribute on the result object they are run with, which will be set if the test run is to be aborted.

The tests are run with an ExtendedToStreamDecorator wrapped around a StreamToQueue instance. ConcurrentStreamTestSuite dequeues events from the queue and forwards them to result. Tests can therefore be either original unittest tests (or compatible tests), or new tests that emit StreamResult events directly.

Parameters **result** – A StreamResult instance. The caller is responsible for calling startTestRun on this instance prior to invoking suite.run, and stopTestRun subsequent to the run method returning.

class testtools.**DecorateTestCaseResult**(*case*, *callout*, *before_run=None*, *after_run=None*)
Decorate a TestCase and permit customisation of the result for runs.

testtools.**ErrorHolder**(*test_id*, *error*, *short_description=None*, *details=None*)
Construct an *ErrorHolder*.

Parameters

- **test_id** – The id of the test.
- **error** – The exc info tuple that will be used as the test's error. This is inserted into the details as 'traceback' - any existing key will be overridden.
- **short_description** – An optional short description of the test.
- **details** – Outcome details as accepted by addSuccess etc.

class testtools.**ExpectedException**(*exc_type*, *value_re=None*, *msg=None*)
A context manager to handle expected exceptions.

def test_foo(self):

with ExpectedException(ValueError, 'fo.*'): raise ValueError('foo')

will pass. If the raised exception has a type other than the specified type, it will be re-raised. If it has a 'str()' that does not match the given regular expression, an AssertionError will be raised. If no exception is raised, an AssertionError will be raised.

class testtools.**ExtendedToOriginalDecorator**(*decorated*)
Permit new TestResult API code to degrade gracefully with old results.

This decorates an existing TestResult and converts missing outcomes such as addSkip to older outcomes such as addSuccess. It also supports the extended details protocol. In all cases the most recent protocol is attempted first, and fallbacks only occur when the decorated result does not support the newer style of calling.

class testtools.**ExtendedToStreamDecorator**(*decorated*)
Permit using old TestResult API code with new StreamResult objects.

This decorates a StreamResult and converts old (Python 2.6 / 2.7 / Extended) TestResult API calls into StreamResult calls.

It also supports regular StreamResult calls, making it safe to wrap around any StreamResult.

current_tags
The currently set tags.

tags(*new_tags*, *gone_tags*)
Add and remove tags from the test.

Parameters

- **new_tags** – A set of tags to be added to the stream.
- **gone_tags** – A set of tags to be removed from the stream.

testtools.**iterate_tests**(*test_suite_or_case*)
Iterate through all of the test cases in 'test_suite_or_case'.

exception testtools.**MultipleExceptions**
Represents many exceptions raised from some operation.

Variables **args** – The sys.exc_info() tuples for each exception.

class testtools.**MultiTestResult**(**results*)
A test result that dispatches to many test results.

wasSuccessful()
Was this result successful?

Only returns True if every constituent result was successful.

class testtools.**PlaceHolder**(*test_id*, *short_description=None*, *details=None*, *outcome='addSuccess'*, *error=None*, *tags=None*, *timestamps=(None*, *None)*)
A placeholder test.

PlaceHolder implements much of the same interface as TestCase and is particularly suitable for being added to TestResults.

testtools.**run_test_with**(*test_runner*, ***kwargs*)
Decorate a test as using a specific RunTest.

e.g.:

```
@run_test_with(CustomRunner, timeout=42)
def test_foo(self):
    self.assertTrue(True)
```

The returned decorator works by setting an attribute on the decorated function. *TestCase.__init__* looks for this attribute when deciding on a RunTest factory. If you wish to use multiple decorators on a test method, then you must either make this one the top-most decorator, or you must write your decorators so that they update the wrapping function with the attributes of the wrapped function. The latter is recommended style anyway. functools.wraps, functools.wrapper and twisted.python.util.mergeFunctionMetadata can help you do this.

Parameters

- **test_runner** – A RunTest factory that takes a test case and an optional list of exception handlers. See RunTest.
- **kwargs** – Keyword arguments to pass on as extra arguments to 'test_runner'.

Returns A decorator to be used for marking a test as needing a special runner.

class testtools.**ResourcedToStreamDecorator**(*decorated*)
Report testresources-related activity to StreamResult objects.

Implement the resource lifecycle TestResult protocol extension supported by the testresources.TestResourceManager class. At each stage of a resource's lifecycle, a stream event with relevant details will be emitted.

Each stream event will have its test_id field set to the resource manager's identifier (see `testresources.TestResourceManager.id()`) plus the method being executed (either 'make' or 'clean').

The test_status will be either 'inprogress' or 'success'.

The runnable flag will be set to False.

class `testtools.`**`Tagger`**(*decorated*, *new_tags*, *gone_tags*)
Tag each test individually.

class `testtools.`**`TestCase`**(**args*, ***kwargs*)
Extensions to the basic TestCase.

Variables

- **`exception_handlers`** – Exceptions to catch from setUp, runTest and tearDown. This list is able to be modified at any time and consists of (exception_class, handler(case, result, exception_value)) pairs.
- **`force_failure`** – Force testtools.RunTest to fail the test after the test has completed.
- ***`run_tests_with`*** – A factory to make the `RunTest` to run tests with. Defaults to `RunTest`. The factory is expected to take a test case and an optional list of exception handlers.

`addCleanup`(*function*, **arguments*, ***keywordArguments*)
Add a cleanup function to be called after tearDown.

Functions added with addCleanup will be called in reverse order of adding after tearDown, or after setUp if setUp raises an exception.

If a function added with addCleanup raises an exception, the error will be recorded as a test error, and the next cleanup will then be run.

Cleanup functions are always called before a test finishes running, even if setUp is aborted by an exception.

`addDetail`(*name*, *content_object*)
Add a detail to be reported with this test's outcome.

For more details see pydoc testtools.TestResult.

Parameters

- **`name`** – The name to give this detail.
- **`content_object`** – The content object for this detail. See testtools.content for more detail.

`addDetailUniqueName`(*name*, *content_object*)
Add a detail to the test, but ensure it's name is unique.

This method checks whether `name` conflicts with a detail that has already been added to the test. If it does, it will modify `name` to avoid the conflict.

For more details see pydoc testtools.TestResult.

Parameters

- **`name`** – The name to give this detail.
- **`content_object`** – The content object for this detail. See testtools.content for more detail.

`addOnException`(*handler*)
Add a handler to be called when an exception occurs in test code.

This handler cannot affect what result methods are called, and is called before any outcome is called on the result object. An example use for it is to add some diagnostic state to the test details dict which is expensive to calculate and not interesting for reporting in the success case.

Handlers are called before the outcome (such as addFailure) that the exception has caused.

Handlers are called in first-added, first-called order, and if they raise an exception, that will propogate out of the test running machinery, halting test processing. As a result, do not call code that may unreasonably fail.

assertEqual(*expected*, *observed*, *message=''*)
Assert that 'expected' is equal to 'observed'.

Parameters

- **expected** – The expected value.
- **observed** – The observed value.
- **message** – An optional message to include in the error.

assertEquals(*expected*, *observed*, *message=''*)
Assert that 'expected' is equal to 'observed'.

Parameters

- **expected** – The expected value.
- **observed** – The observed value.
- **message** – An optional message to include in the error.

assertIn(*needle*, *haystack*, *message=''*)
Assert that needle is in haystack.

assertIs(*expected*, *observed*, *message=''*)
Assert that 'expected' is 'observed'.

Parameters

- **expected** – The expected value.
- **observed** – The observed value.
- **message** – An optional message describing the error.

assertIsNone(*observed*, *message=''*)
Assert that 'observed' is equal to None.

Parameters

- **observed** – The observed value.
- **message** – An optional message describing the error.

assertIsNot(*expected*, *observed*, *message=''*)
Assert that 'expected' is not 'observed'.

assertIsNotNone(*observed*, *message=''*)
Assert that 'observed' is not equal to None.

Parameters

- **observed** – The observed value.
- **message** – An optional message describing the error.

assertNotIn(*needle*, *haystack*, *message=''*)
Assert that needle is not in haystack.

assertRaises(*excClass*, *callableObj*, **args*, ***kwargs*)
Fail unless an exception of class excClass is thrown by callableObj when invoked with arguments args and keyword arguments kwargs. If a different type of exception is thrown, it will not be caught, and the test case will be deemed to have suffered an error, exactly as for an unexpected exception.

assertThat(*matchee*, *matcher*, *message=''*, *verbose=False*)
Assert that matchee is matched by matcher.

Parameters

- **matchee** – An object to match with matcher.
- **matcher** – An object meeting the testtools.Matcher protocol.

Raises MismatchError – When matcher does not match thing.

expectFailure(*reason*, *predicate*, **args*, ***kwargs*)
Check that a test fails in a particular way.

If the test fails in the expected way, a KnownFailure is caused. If it succeeds an UnexpectedSuccess is caused.

The expected use of expectFailure is as a barrier at the point in a test where the test would fail. For example: >>> def test_foo(self): >>> self.expectFailure("1 should be 0", self.assertNotEqual, 1, 0) >>> self.assertEqual(1, 0)

If in the future 1 were to equal 0, the expectFailure call can simply be removed. This separation preserves the original intent of the test while it is in the expectFailure mode.

expectThat(*matchee*, *matcher*, *message=''*, *verbose=False*)
Check that matchee is matched by matcher, but delay the assertion failure.

This method behaves similarly to `assertThat`, except that a failed match does not exit the test immediately. The rest of the test code will continue to run, and the test will be marked as failing after the test has finished.

Parameters

- **matchee** – An object to match with matcher.
- **matcher** – An object meeting the testtools.Matcher protocol.
- **message** – If specified, show this message with any failed match.

failUnlessEqual(*expected*, *observed*, *message=''*)
Assert that 'expected' is equal to 'observed'.

Parameters

- **expected** – The expected value.
- **observed** – The observed value.
- **message** – An optional message to include in the error.

failUnlessRaises(*excClass*, *callableObj*, **args*, ***kwargs*)
Fail unless an exception of class excClass is thrown by callableObj when invoked with arguments args and keyword arguments kwargs. If a different type of exception is thrown, it will not be caught, and the test case will be deemed to have suffered an error, exactly as for an unexpected exception.

getDetails()
Get the details dict that will be reported with this test's outcome.

For more details see pydoc testtools.TestResult.

getUniqueInteger()
Get an integer unique to this test.

Returns an integer that is guaranteed to be unique to this instance. Use this when you need an arbitrary integer in your test, or as a helper for custom anonymous factory methods.

getUniqueString(*prefix=None*)
Get a string unique to this test.

Returns a string that is guaranteed to be unique to this instance. Use this when you need an arbitrary string in your test, or as a helper for custom anonymous factory methods.

Parameters **prefix** – The prefix of the string. If not provided, defaults to the id of the tests.

Returns A bytestring of '<prefix>-<unique_int>'.

onException(*exc_info*, *tb_label='traceback'*)
Called when an exception propagates from test code.

Seealso addOnException

patch(*obj*, *attribute*, *value*)
Monkey-patch 'obj.attribute' to 'value' while the test is running.

If 'obj' has no attribute, then the monkey-patch will still go ahead, and the attribute will be deleted instead of restored to its original value.

Parameters

- **obj** – The object to patch. Can be anything.
- **attribute** – The attribute on 'obj' to patch.
- **value** – The value to set 'obj.attribute' to.

run_tests_with
alias of *RunTest*

skip(*reason*)
DEPRECATED: Use skipTest instead.

skipException
alias of SkipTest

skipTest(*reason*)
Cause this test to be skipped.

This raises self.skipException(reason). skipException is raised to permit a skip to be triggered at any point (during setUp or the testMethod itself). The run() method catches skipException and translates that into a call to the result objects addSkip method.

Parameters **reason** – The reason why the test is being skipped. This must support being cast into a unicode string for reporting.

useFixture(*fixture*)
Use fixture in a test case.

The fixture will be setUp, and self.addCleanup(fixture.cleanUp) called.

Parameters **fixture** – The fixture to use.

Returns The fixture, after setting it up and scheduling a cleanup for it.

class testtools.**TestCommand**(*dist*)
Command to run unit tests with testtools

class testtools.**TestByTestResult**(*on_test*)
Call something every time a test completes.

class testtools.**TestResult**(*failfast=False*, *tb_locals=False*)
Subclass of unittest.TestResult extending the protocol for flexibility.

This test result supports an experimental protocol for providing additional data to in test outcomes. All the outcome methods take an optional dict 'details'. If supplied any other detail parameters like 'err' or 'reason' should not be provided. The details dict is a mapping from names to MIME content objects (see testtools.content). This permits attaching tracebacks, log files, or even large objects like databases that were part of the test fixture. Until this API is accepted into upstream Python it is considered experimental: it may be replaced at any point by a newer version more in line with upstream Python. Compatibility would be aimed for in this case, but may not be possible.

Variables **skip_reasons** – A dict of skip-reasons -> list of tests. See addSkip.

addError(*test*, *err=None*, *details=None*)
Called when an error has occurred. 'err' is a tuple of values as returned by sys.exc_info().

Parameters **details** – Alternative way to supply details about the outcome. see the class docstring for more information.

addExpectedFailure(*test*, *err=None*, *details=None*)
Called when a test has failed in an expected manner.

Like with addSuccess and addError, testStopped should still be called.

Parameters

- **test** – The test that has been skipped.
- **err** – The exc_info of the error that was raised.

Returns None

addFailure(*test*, *err=None*, *details=None*)
Called when an error has occurred. 'err' is a tuple of values as returned by sys.exc_info().

Parameters **details** – Alternative way to supply details about the outcome. see the class docstring for more information.

addSkip(*test*, *reason=None*, *details=None*)
Called when a test has been skipped rather than running.

Like with addSuccess and addError, testStopped should still be called.

This must be called by the TestCase. 'addError' and 'addFailure' will not call addSkip, since they have no assumptions about the kind of errors that a test can raise.

Parameters

- **test** – The test that has been skipped.
- **reason** – The reason for the test being skipped. For instance, u"pyGL is not available".
- **details** – Alternative way to supply details about the outcome. see the class docstring for more information.

Returns None

addSuccess(*test*, *details=None*)
Called when a test succeeded.

addUnexpectedSuccess(*test*, *details=None*)
Called when a test was expected to fail, but succeed.

current_tags
The currently set tags.

done()
Called when the test runner is done.

deprecated in favour of stopTestRun.

startTestRun()
Called before a test run starts.

New in Python 2.7. The testtools version resets the result to a pristine condition ready for use in another test run. Note that this is different from Python 2.7's startTestRun, which does nothing.

stopTestRun()
Called after a test run completes

New in python 2.7

tags(*new_tags*, *gone_tags*)
Add and remove tags from the test.

Parameters

- **new_tags** – A set of tags to be added to the stream.
- **gone_tags** – A set of tags to be removed from the stream.

time(*a_datetime*)
Provide a timestamp to represent the current time.

This is useful when test activity is time delayed, or happening concurrently and getting the system time between API calls will not accurately represent the duration of tests (or the whole run).

Calling time() sets the datetime used by the TestResult object. Time is permitted to go backwards when using this call.

Parameters **a_datetime** – A datetime.datetime object with TZ information or None to reset the TestResult to gathering time from the system.

wasSuccessful()
Has this result been successful so far?

If there have been any errors, failures or unexpected successes, return False. Otherwise, return True.

Note: This differs from standard unittest in that we consider unexpected successes to be equivalent to failures, rather than successes.

class testtools.**TestResultDecorator**(*decorated*)
General pass-through decorator.

This provides a base that other TestResults can inherit from to gain basic forwarding functionality.

class testtools.**TextTestResult**(*stream*, *failfast=False*, *tb_locals=False*)
A TestResult which outputs activity to a text stream.

class testtools.**RunTest**(*case*, *handlers=None*, *last_resort=None*)
An object to run a test.

RunTest objects are used to implement the internal logic involved in running a test. TestCase.__init__ stores _RunTest as the class of RunTest to execute. Passing the runTest= parameter to TestCase.__init__ allows a different RunTest class to be used to execute the test.

Subclassing or replacing RunTest can be useful to add functionality to the way that tests are run in a given project.

Variables

- **case** – The test case that is to be run.
- **result** – The result object a case is reporting to.
- **handlers** – A list of (ExceptionClass, handler_function) for exceptions that should be caught if raised from the user code. Exceptions that are caught are checked against this list in first to last order. There is a catch-all of 'Exception' at the end of the list, so to add a new exception to the list, insert it at the front (which ensures that it will be checked before any existing base classes in the list. If you add multiple exceptions some of which are subclasses of each other, add the most specific exceptions last (so they come before their parent classes in the list).
- **exception_caught** – An object returned when _run_user catches an exception.
- **_exceptions** – A list of caught exceptions, used to do the single reporting of error/failure/skip etc.

run(*result=None*)
Run self.case reporting activity to result.

Parameters **result** – Optional testtools.TestResult to report activity to.

Returns The result object the test was run against.

testtools.**skip**(*reason*)
A decorator to skip unit tests.

This is just syntactic sugar so users don't have to change any of their unit tests in order to migrate to python 2.7, which provides the @unittest.skip decorator.

testtools.**skipIf**(*condition*, *reason*)
A decorator to skip a test if the condition is true.

testtools.**skipUnless**(*condition*, *reason*)
A decorator to skip a test unless the condition is true.

class testtools.**StreamFailFast**(*on_error*)
Call the supplied callback if an error is seen in a stream.

An example callback:

```
def do_something():
    pass
```

class testtools.**StreamResult**
A test result for reporting the activity of a test run.

Typical use

```
>>> result = StreamResult()
>>> result.startTestRun()
>>> try:
...     case.run(result)
... finally:
...     result.stopTestRun()
```

The case object will be either a TestCase or a TestSuite, and generally make a sequence of calls like:

```
>>> result.status(self.id(), 'inprogress')
>>> result.status(self.id(), 'success')
```

General concepts

StreamResult is built to process events that are emitted by tests during a test run or test enumeration. The test run may be running concurrently, and even be spread out across multiple machines.

All events are timestamped to prevent network buffering or scheduling latency causing false timing reports. Timestamps are datetime objects in the UTC timezone.

A route_code is a unicode string that identifies where a particular test run. This is optional in the API but very useful when multiplexing multiple streams together as it allows identification of interactions between tests that were run on the same hardware or in the same test process. Generally actual tests never need to bother with this - it is added and processed by StreamResult's that do multiplexing / run analysis. route_codes are also used to route stdin back to pdb instances.

The StreamResult base class does no accounting or processing, rather it just provides an empty implementation of every method, suitable for use as a base class regardless of intent.

startTestRun()
Start a test run.

This will prepare the test result to process results (which might imply connecting to a database or remote machine).

status(*test_id=None*, *test_status=None*, *test_tags=None*, *runnable=True*, *file_name=None*, *file_bytes=None*, *eof=False*, *mime_type=None*, *route_code=None*, *timestamp=None*)
Inform the result about a test status.

Parameters

- **test_id** – The test whose status is being reported. None to report status about the test run as a whole.
- **test_status** – The status for the test. There are two sorts of status - interim and final status events. As many interim events can be generated as desired, but only one final event. After a final status event any further file or status events from the same test_id+route_code may be discarded or associated with a new test by the StreamResult. (But no exception will be thrown).

  Interim states:

  - None - no particular status is being reported, or status being reported is not associated with a test (e.g. when reporting on stdout / stderr chatter).
  - inprogress - the test is currently running. Emitted by tests when they start running and at any intermediary point they might choose to indicate their continual operation.

  Final states:

  - exists - the test exists. This is used when a test is not being executed. Typically this is when querying what tests could be run in a test run (which is useful for selecting tests to run).
  - xfail - the test failed but that was expected. This is purely informative - the test is not considered to be a failure.
  - uxsuccess - the test passed but was expected to fail. The test will be considered a failure.
  - success - the test has finished without error.

  - fail - the test failed (or errored). The test will be considered a failure.
  - skip - the test was selected to run but chose to be skipped. e.g. a test dependency was missing. This is purely informative- the test is not considered to be a failure.

- **test_tags** – Optional set of tags to apply to the test. Tags have no intrinsic meaning - that is up to the test author.
- **runnable** – Allows status reports to mark that they are for tests which are not able to be explicitly run. For instance, subtests will report themselves as non-runnable.
- **file_name** – The name for the file_bytes. Any unicode string may be used. While there is no semantic value attached to the name of any attachment, the names 'stdout' and 'stderr' and 'traceback' are recommended for use only for output sent to stdout, stderr and tracebacks of exceptions. When file_name is supplied, file_bytes must be a bytes instance.
- **file_bytes** – A bytes object containing content for the named file. This can just be a single chunk of the file - emitting another file event with more later. Must be None unleses a file_name is supplied.
- **eof** – True if this chunk is the last chunk of the file, any additional chunks with the same name should be treated as an error and discarded. Ignored unless file_name has been supplied.
- **mime_type** – An optional MIME type for the file. stdout and stderr will generally be "text/plain; charset=utf8". If None, defaults to application/octet-stream. Ignored unless file_name has been supplied.

stopTestRun()
Stop a test run.

This informs the result that no more test updates will be received. At this point any test ids that have started and not completed can be considered failed-or-hung.

class testtools.**StreamResultRouter**(*fallback=None*, *do_start_stop_run=True*)
A StreamResult that routes events.

StreamResultRouter forwards received events to another StreamResult object, selected by a dynamic forwarding policy. Events where no destination is found are forwarded to the fallback StreamResult, or an error is raised.

Typical use is to construct a router with a fallback and then either create up front mapping rules, or create them as-needed from the fallback handler:

```
>>> router = StreamResultRouter()
>>> sink = doubles.StreamResult()
>>> router.add_rule(sink, 'route_code_prefix', route_prefix='0',
...     consume_route=True)
>>> router.status(
...     test_id='foo', route_code='0/1', test_status='uxsuccess')
```

StreamResultRouter has no buffering.

When adding routes (and for the fallback) whether to call startTestRun and stopTestRun or to not call them is controllable by passing 'do_start_stop_run'. The default is to call them for the fallback only. If a route is added after startTestRun has been called, and do_start_stop_run is True then startTestRun is called immediately on the new route sink.

There is no a-priori defined lookup order for routes: if they are ambiguous the behaviour is undefined. Only a single route is chosen for any event.

add_rule(*sink*, *policy*, *do_start_stop_run=False*, ***policy_args*)
Add a rule to route events to sink when they match a given policy.

Parameters

- **sink** – A StreamResult to receive events.
- **policy** – A routing policy. Valid policies are 'route_code_prefix' and 'test_id'.
- **do_start_stop_run** – If True then startTestRun and stopTestRun events will be passed onto this sink.

Raises ValueError if the policy is unknown

Raises TypeError if the policy is given arguments it cannot handle.

`route_code_prefix` routes events based on a prefix of the route code in the event. It takes a `route_prefix` argument to match on (e.g. '0') and a `consume_route` argument, which, if True, removes the prefix from the `route_code` when forwarding events.

`test_id` routes events based on the test id. It takes a single argument, `test_id`. Use `None` to select non-test events.

class testtools.**StreamSummary**
A specialised StreamResult that summarises a stream.

The summary uses the same representation as the original unittest.TestResult contract, allowing it to be consumed by any test runner.

wasSuccessful()
Return False if any failure has occured.

Note that incomplete tests can only be detected when stopTestRun is called, so that should be called before checking wasSuccessful.

class testtools.**StreamTagger**(*targets*, *add=None*, *discard=None*)
Adds or discards tags from StreamResult events.

class testtools.**StreamToDict**(*on_test*)
A specialised StreamResult that emits a callback as tests complete.

Top level file attachments are simply discarded. Hung tests are detected by stopTestRun and notified there and then.

The callback is passed a dict with the following keys:

•id: the test id.

•tags: The tags for the test. A set of unicode strings.

•details: A dict of file attachments - `testtools.content.Content` objects.

•status: One of the StreamResult status codes (including inprogress) or 'unknown' (used if only file events for a test were received...)

•timestamps: A pair of timestamps - the first one received with this test id, and the one in the event that triggered the notification. Hung tests have a None for the second end event. Timestamps are not compared - their ordering is purely order received in the stream.

Only the most recent tags observed in the stream are reported.

class testtools.**StreamToExtendedDecorator**(*decorated*)
Convert StreamResult API calls into ExtendedTestResult calls.

This will buffer all calls for all concurrently active tests, and then flush each test as they complete.

Incomplete tests will be flushed as errors when the test run stops.

Non test file attachments are accumulated into a test called 'testtools.extradata' flushed at the end of the run.

class testtools.**StreamToQueue**(*queue*, *routing_code*)
A StreamResult which enqueues events as a dict to a queue.Queue.

Events have their route code updated to include the route code StreamToQueue was constructed with before they are submitted. If the event route code is None, it is replaced with the StreamToQueue route code, otherwise it is prefixed with the supplied code + a hyphen.

startTestRun and stopTestRun are forwarded to the queue. Implementors that dequeue events back into StreamResult calls should take care not to call startTestRun / stopTestRun on other StreamResult objects multiple times (e.g. by filtering startTestRun and stopTestRun).

`StreamToQueue` is typically used by `ConcurrentStreamTestSuite`, which creates one `StreamToQueue` per thread, forwards status events to the the StreamResult that `ConcurrentStreamTestSuite.run()` was called with, and uses the stopTestRun event to trigger calling join() on the each thread.

Unlike ThreadsafeForwardingResult which this supercedes, no buffering takes place - any event supplied to a StreamToQueue will be inserted into the queue immediately.

Events are forwarded as a dict with a key `event` which is one of `startTestRun`, `stopTestRun` or `status`. When `event` is `status` the dict also has keys matching the keyword arguments of `StreamResult.status`, otherwise it has one other key `result` which is the result that invoked `startTestRun`.

route_code(*route_code*)
Adjust route_code on the way through.

class testtools.**TestControl**
Controls a running test run, allowing it to be interrupted.

Variables **shouldStop** – If True, tests should not run and should instead return immediately. Similarly a TestSuite should check this between each test and if set stop dispatching any new tests and return.

stop()
Indicate that tests should stop running.

class testtools.**ThreadsafeForwardingResult**(*target*, *semaphore*)
A TestResult which ensures the target does not receive mixed up calls.

Multiple `ThreadsafeForwardingResults` can forward to the same target result, and that target result will only ever receive the complete set of events for one test at a time.

This is enforced using a semaphore, which further guarantees that tests will be sent atomically even if the `ThreadsafeForwardingResults` are in different threads.

`ThreadsafeForwardingResult` is typically used by `ConcurrentTestSuite`, which creates one `ThreadsafeForwardingResult` per thread, each of which wraps of the TestResult that `ConcurrentTestSuite.run()` is called with.

target.startTestRun() and target.stopTestRun() are called once for each ThreadsafeForwardingResult that forwards to the same target. If the target takes special action on these events, it should take care to accommodate this.

time() and tags() calls are batched to be adjacent to the test result and in the case of tags() are coerced into test-local scope, avoiding the opportunity for bugs around global state in the target.

tags(*new_tags*, *gone_tags*)
See *TestResult*.

class testtools.**TimestampingStreamResult**(*target*)
A StreamResult decorator that assigns a timestamp when none is present.

This is convenient for ensuring events are timestamped.

testtools.**try_import**(*name*, *alternative=None*, *error_callback=None*)
Attempt to import `name`. If it fails, return `alternative`.

When supporting multiple versions of Python or optional dependencies, it is useful to be able to try to import a module.

Parameters

- **name** – The name of the object to import, e.g. `os.path` or `os.path.join`.
- **alternative** – The value to return if no module can be imported. Defaults to None.
- **error_callback** – If non-None, a callable that is passed the ImportError when the module cannot be loaded.

testtools.**try_imports**(*module_names*, *alternative=<object object>*, *error_callback=None*)
Attempt to import modules.

Tries to import the first module in `module_names`. If it can be imported, we return it. If not, we go on to the second module and try that. The process continues until we run out of modules to try. If none of the modules can be imported, either raise an exception or return the provided `alternative` value.

Parameters

- **module_names** – A sequence of module names to try to import.
- **alternative** – The value to return if no module can be imported. If unspecified, we raise an ImportError.
- **error_callback** – If None, called with the ImportError for *each* module that fails to load.

Raises **ImportError** – If none of the modules can be imported and no alternative value was specified.

testtools.**unique_text_generator**(*prefix*)
Generates unique text values.

Generates text values that are unique. Use this when you need arbitrary text in your test, or as a helper for custom anonymous factory methods.

Parameters **prefix** – The prefix for text.

Returns text that looks like '<prefix>-<text_with_unicode>'.

Return type six.text_type

testtools.assertions

Assertion helpers.

testtools.assertions.**assert_that**(*matchee*, *matcher*, *message=''*, *verbose=False*)
Assert that matchee is matched by matcher.

This should only be used when you need to use a function based matcher, assertThat in Testtools.Testcase is prefered and has more features

Parameters

- **matchee** – An object to match with matcher.
- **matcher** – An object meeting the testtools.Matcher protocol.

Raises **MismatchError** – When matcher does not match thing.

testtools.matchers

All the matchers.

Matchers, a way to express complex assertions outside the testcase.

Inspired by 'hamcrest'.

Matcher provides the abstract API that all matchers need to implement.

Bundled matchers are listed in __all__: a list can be obtained by running $ python -c 'import testtools.matchers; print testtools.matchers.__all__'

class testtools.matchers.**AfterPreprocessing**(*preprocessor*, *matcher*, *annotate=True*)
Matches if the value matches after passing through a function.

This can be used to aid in creating trivial matchers as functions, for example:

```
def PathHasFileContent(content):
    def _read(path):
        return open(path).read()
    return AfterPreprocessing(_read, Equals(content))
```

class testtools.matchers.**AllMatch**(*matcher*)
Matches if all provided values match the given matcher.

testtools.matchers.**Always**()
Always match.

That is:

```
self.assertThat(x, Always())
```

Will always match and never fail, no matter what x is. Most useful when passed to other higher-order matchers (e.g. *MatchesListwise*).

class testtools.matchers.**Annotate**(*annotation*, *matcher*)
Annotates a matcher with a descriptive string.

Mismatches are then described as '<mismatch>: <annotation>'.

classmethod **if_message**(*annotation*, *matcher*)
Annotate matcher only if annotation is non-empty.

class testtools.matchers.**AnyMatch**(*matcher*)
Matches if any of the provided values match the given matcher.

class testtools.matchers.**Contains**(*needle*)
Checks whether something is contained in another thing.

testtools.matchers.**ContainsAll**(*items*)
Make a matcher that checks whether a list of things is contained in another thing.

The matcher effectively checks that the provided sequence is a subset of the matchee.

class testtools.matchers.**ContainedByDict**(*expected*)
Match a dictionary for which this is a super-dictionary.

Specify a dictionary mapping keys (often strings) to matchers. This is the 'expected' dict. Any dictionary that matches this must have **only** these keys, and the values must match the corresponding matchers in the expected dict. Dictionaries that have fewer keys can also match.

In other words, any matching dictionary must be contained by the dictionary given to the constructor.

Does not check for strict super-dictionary. That is, equal dictionaries match.

class testtools.matchers.**ContainsDict**(*expected*)
Match a dictionary for that contains a specified sub-dictionary.

Specify a dictionary mapping keys (often strings) to matchers. This is the 'expected' dict. Any dictionary that matches this must have **at least** these keys, and the values must match the corresponding matchers in the expected dict. Dictionaries that have more keys will also match.

In other words, any matching dictionary must contain the dictionary given to the constructor.

Does not check for strict sub-dictionary. That is, equal dictionaries match.

class testtools.matchers.**DirContains**(*filenames=None*, *matcher=None*)
Matches if the given directory contains files with the given names.

That is, is the directory listing exactly equal to the given files?

testtools.matchers.**DirExists**()
Matches if the path exists and is a directory.

class testtools.matchers.**DocTestMatches**(*example*, *flags=0*)
See if a string matches a doctest example.

class testtools.matchers.**EndsWith**(*expected*)
Checks whether one string ends with another.

class testtools.matchers.**Equals**(*expected*)
Matches if the items are equal.

comparator()
eq(a, b) – Same as a==b.

class testtools.matchers.**FileContains**(*contents=None*, *matcher=None*)
Matches if the given file has the specified contents.

testtools.matchers.**FileExists**()
Matches if the given path exists and is a file.

class testtools.matchers.**GreaterThan**(*expected*)
Matches if the item is greater than the matchers reference object.

comparator()
gt(a, b) – Same as a>b.

class testtools.matchers.**HasPermissions**(*octal_permissions*)
Matches if a file has the given permissions.

Permissions are specified and matched as a four-digit octal string.

class testtools.matchers.**Is**(*expected*)
Matches if the items are identical.

comparator()
is_(a, b) – Same as a is b.

testtools.matchers.**IsDeprecated**(*message*)
Make a matcher that checks that a callable produces exactly one *DeprecationWarning*.

Parameters **message** – Matcher for the warning message.

class testtools.matchers.**IsInstance**(**types*)
Matcher that wraps isinstance.

class testtools.matchers.**KeysEqual**(**expected*)
Checks whether a dict has particular keys.

class testtools.matchers.**LessThan**(*expected*)
Matches if the item is less than the matchers reference object.

comparator()
lt(a, b) – Same as a<b.

class testtools.matchers.**MatchesAll**(**matchers*, ***options*)
Matches if all of the matchers it is created with match.

class testtools.matchers.**MatchesAny**(**matchers*)
Matches if any of the matchers it is created with match.

class testtools.matchers.**MatchesDict**(*expected*)
Match a dictionary exactly, by its keys.

Specify a dictionary mapping keys (often strings) to matchers. This is the 'expected' dict. Any dictionary that matches this must have exactly the same keys, and the values must match the corresponding matchers in the expected dict.

class testtools.matchers.**MatchesException**(*exception*, *value_re=None*)
Match an exc_info tuple against an exception instance or type.

class testtools.matchers.**MatchesListwise**(*matchers*, *first_only=False*)
Matches if each matcher matches the corresponding value.

More easily explained by example than in words:

```
>>> from ._basic import Equals
>>> MatchesListwise([Equals(1)]).match([1])
>>> MatchesListwise([Equals(1), Equals(2)]).match([1, 2])
>>> print (MatchesListwise([Equals(1), Equals(2)]).match([2, 1]).describe())
Differences: [
2 != 1
1 != 2
]
>>> matcher = MatchesListwise([Equals(1), Equals(2)], first_only=True)
>>> print (matcher.match([3, 4]).describe())
3 != 1
```

class testtools.matchers.**MatchesPredicate**(*predicate*, *message*)
Match if a given function returns True.

It is reasonably common to want to make a very simple matcher based on a function that you already have that returns True or False given a single argument (i.e. a predicate function). This matcher makes it very easy to do so. e.g.:

```
IsEven = MatchesPredicate(lambda x: x % 2 == 0, '%s is not even')
self.assertThat(4, IsEven)
```

testtools.matchers.**MatchesPredicateWithParams**(*predicate*, *message*, *name=None*)
Match if a given parameterised function returns True.

It is reasonably common to want to make a very simple matcher based on a function that you already have that returns True or False given some arguments. This matcher makes it very easy to do so. e.g.:

```
HasLength = MatchesPredicate(
    lambda x, y: len(x) == y, 'len({0}) is not {1}')
# This assertion will fail, as 'len([1, 2]) == 3' is False.
self.assertThat([1, 2], HasLength(3))
```

Note that unlike MatchesPredicate MatchesPredicateWithParams returns a factory which you then customise to use by constructing an actual matcher from it.

The predicate function should take the object to match as its first parameter. Any additional parameters supplied when constructing a matcher are supplied to the predicate as additional parameters when checking for a match.

Parameters

- **predicate** – The predicate function.
- **message** – A format string for describing mis-matches.
- **name** – Optional replacement name for the matcher.

class testtools.matchers.**MatchesRegex**(*pattern*, *flags=0*)
Matches if the matchee is matched by a regular expression.

class testtools.matchers.**MatchesSetwise**(**matchers*)
Matches if all the matchers match elements of the value being matched.

That is, each element in the 'observed' set must match exactly one matcher from the set of matchers, with no matchers left over.

The difference compared to *MatchesListwise* is that the order of the matchings does not matter.

class testtools.matchers.**MatchesStructure**(***kwargs*)
Matcher that matches an object structurally.

'Structurally' here means that attributes of the object being matched are compared against given matchers.

fromExample allows the creation of a matcher from a prototype object and then modified versions can be created with *update*.

byEquality creates a matcher in much the same way as the constructor, except that the matcher for each of the attributes is assumed to be *Equals*.

byMatcher creates a similar matcher to *byEquality*, but you get to pick the matcher, rather than just using *Equals*.

classmethod byEquality(***kwargs*)
Matches an object where the attributes equal the keyword values.

Similar to the constructor, except that the matcher is assumed to be Equals.

classmethod byMatcher(*matcher*, ***kwargs*)
Matches an object where the attributes match the keyword values.

Similar to the constructor, except that the provided matcher is used to match all of the values.

testtools.matchers.**Never**()
Never match.

That is:

```
self.assertThat(x, Never())
```

Will never match and always fail, no matter what `x` is. Included for completeness with *`Always()`*, but if you find a use for this, let us know!

class testtools.matchers.**NotEquals**(*expected*)
Matches if the items are not equal.

In most cases, this is equivalent to `Not(Equals(foo))`. The difference only matters when testing `__ne__` implementations.

comparator()
ne(a, b) – Same as a!=b.

class testtools.matchers.**Not**(*matcher*)
Inverts a matcher.

testtools.matchers.**PathExists**()
Matches if the given path exists.

Use like this:

```
assertThat('/some/path', PathExists())
```

class testtools.matchers.**Raises**(*exception_matcher=None*)
Match if the matchee raises an exception when called.

Exceptions which are not subclasses of Exception propogate out of the Raises.match call unless they are explicitly matched.

testtools.matchers.**raises**(*exception*)
Make a matcher that checks that a callable raises an exception.

This is a convenience function, exactly equivalent to:

```
return Raises(MatchesException(exception))
```

See *Raises* and *MatchesException* for more information.

class testtools.matchers.**SamePath**(*path*)
Matches if two paths are the same.

That is, the paths are equal, or they point to the same file but in different ways. The paths do not have to exist.

class testtools.matchers.**StartsWith**(*expected*)
Checks whether one string starts with another.

class testtools.matchers.**TarballContains**(*paths*)
Matches if the given tarball contains the given paths.

Uses TarFile.getnames() to get the paths out of the tarball.

class testtools.matchers.**Warnings**(*warnings_matcher=None*)
Match if the matchee produces warnings.

testtools.matchers.**WarningMessage**(*category_type*, *message=None*, *filename=None*, *lineno=None*, *line=None*)
Create a matcher that will match *warnings.WarningMessage*s.

For example, to match captured *DeprecationWarning*s with a message about some `foo` being replaced with `bar`:

```
WarningMessage(DeprecationWarning,
               message=MatchesAll(
                   Contains('foo is deprecated'),
                   Contains('use bar instead')))
```

Parameters **category_type** (*type*) – A warning type, for example

DeprecationWarning. :param message_matcher: A matcher object that will be evaluated against warning's message. :param filename_matcher: A matcher object that will be evaluated against the warning's filename. :param lineno_matcher: A matcher object that will be evaluated against the warning's line number. :param line_matcher: A matcher object that will be evaluated against the warning's line of source code.

testtools.twistedsupport

Support for testing code that uses Twisted.

`testtools.twistedsupport.`**`succeeded`**(*matcher*)

Match a Deferred that has fired successfully.

For example:

```
fires_with_the_answer = succeeded(Equals(42))
deferred = defer.succeed(42)
assert_that(deferred, fires_with_the_answer)
```

This assertion will pass. However, if `deferred` had fired with a different value, or had failed, or had not fired at all, then it would fail.

Use this instead of `twisted.trial.unittest.SynchronousTestCase.successResultOf()`.

Parameters **`matcher`** – A matcher to match against the result of a `Deferred`.

Returns A matcher that can be applied to a synchronous `Deferred`.

`testtools.twistedsupport.`**`failed`**(*matcher*)

Match a Deferred that has failed.

For example:

```
error = RuntimeError('foo')
fails_at_runtime = failed(
    AfterPreprocessing(lambda f: f.value, Equals(error)))
deferred = defer.fail(error)
assert_that(deferred, fails_at_runtime)
```

This assertion will pass. However, if `deferred` had fired successfully, had failed with a different error, or had not fired at all, then it would fail.

Use this instead of `twisted.trial.unittest.SynchronousTestCase.failureResultOf()`.

Parameters **`matcher`** – A matcher to match against the result of a failing `Deferred`.

Returns A matcher that can be applied to a synchronous `Deferred`.

`testtools.twistedsupport.`**`has_no_result`**()

Match a Deferred that has not yet fired.

For example, this will pass:

```
assert_that(defer.Deferred(), has_no_result())
```

But this will fail:

```
>>> assert_that(defer.succeed(None), has_no_result())
Traceback (most recent call last):
  ...
  File "testtools/assertions.py", line 22, in assert_that
```

```
    raise MismatchError(matchee, matcher, mismatch, verbose)
testtools.matchers._impl.MismatchError: No result expected on <Deferred at ...␣
↪current result: None>, found None instead
```

As will this:

```
>>> assert_that(defer.fail(RuntimeError('foo')), has_no_result())
Traceback (most recent call last):
  ...
  File "testtools/assertions.py", line 22, in assert_that
    raise MismatchError(matchee, matcher, mismatch, verbose)
testtools.matchers._impl.MismatchError: No result expected on <Deferred at ...␣
↪current result: <twisted.python.failure.Failure <type 'exceptions.RuntimeError'>
↪>>, found <twisted.python.failure.Failure <type 'exceptions.RuntimeError'>>␣
↪instead
```

class testtools.twistedsupport.**AsynchronousDeferredRunTest**(*case*, *handlers=None*, *last_resort=None*, *reactor=None*, *timeout=0.005*, *debug=False*, *suppress_twisted_logging=True*, *store_twisted_logs=True*)

Runner for tests that return Deferreds that fire asynchronously.

Use this runner when you have tests that return Deferreds that will only fire if the reactor is left to spin for a while.

__init__(*case*, *handlers=None*, *last_resort=None*, *reactor=None*, *timeout=0.005*, *debug=False*, *suppress_twisted_logging=True*, *store_twisted_logs=True*)

Construct an AsynchronousDeferredRunTest.

Please be sure to always use keyword syntax, not positional, as the base class may add arguments in future - and for core code compatibility with that we have to insert them before the local parameters.

Parameters

- **case** (TestCase) – The *TestCase* to run.
- **handlers** – A list of exception handlers (ExceptionType, handler) where 'handler' is a callable that takes a *TestCase*, a testtools.TestResult and the exception raised.
- **last_resort** – Handler to call before re-raising uncatchable exceptions (those for which there is no handler).
- **reactor** – The Twisted reactor to use. If not given, we use the default reactor.
- **timeout** (*float*) – The maximum time allowed for running a test. The default is 0.005s.
- **debug** – Whether or not to enable Twisted's debugging. Use this to get information about unhandled Deferreds and left-over DelayedCalls. Defaults to False.
- **suppress_twisted_logging** (*bool*) – If True, then suppress Twisted's default logging while the test is being run. Defaults to True.
- **store_twisted_logs** (*bool*) – If True, then store the Twisted logs that took place during the run as the 'twisted-log' detail. Defaults to True.

classmethod make_factory(*reactor=None*, *timeout=0.005*, *debug=False*, *suppress_twisted_logging=True*, *store_twisted_logs=True*)

Make a factory that conforms to the RunTest factory interface.

Example:

```
class SomeTests(TestCase):
    # Timeout tests after two minutes.
    run_tests_with = AsynchronousDeferredRunTest.make_factory(
        timeout=120)
```

class testtools.twistedsupport.**AsynchronousDeferredRunTestForBrokenTwisted**(*case*, *handlers=None*, *last_resort=None*, *reactor=None*, *timeout=0.005*, *debug=False*, *suppress_twisted_logging=Tr*
store_twisted_logs=True)

Test runner that works around Twisted brokenness re reactor junk.

There are many APIs within Twisted itself where a Deferred fires but leaves cleanup work scheduled for the reactor to do. Arguably, many of these are bugs. This runner iterates the reactor event loop a number of times after every test, in order to shake out these buggy-but-commonplace events.

class testtools.twistedsupport.**SynchronousDeferredRunTest**(*case*, *handlers=None*, *last_resort=None*)

Runner for tests that return synchronous Deferreds.

This runner doesn't touch the reactor at all. It assumes that tests return Deferreds that have already fired.

class testtools.twistedsupport.**CaptureTwistedLogs**

Capture all the Twisted logs and add them as a detail.

Much of the time, you won't need to use this directly, as *AsynchronousDeferredRunTest* captures Twisted logs when the `store_twisted_logs` is set to `True` (which it is by default).

However, if you want to do custom processing of Twisted's logs, then this class can be useful.

For example:

```
class TwistedTests(TestCase):
    run_tests_with(
        partial(AsynchronousDeferredRunTest, store_twisted_logs=False))

    def setUp(self):
        super(TwistedTests, self).setUp()
        twisted_logs = self.useFixture(CaptureTwistedLogs())
        # ... do something with twisted_logs ...
```

testtools.twistedsupport.**assert_fails_with**(*d*, **exc_types*, ***kwargs*)

Assert that `d` will fail with one of `exc_types`.

The normal way to use this is to return the result of `assert_fails_with` from your unit test.

Equivalent to Twisted's `assertFailure`.

Parameters

- **d** (*Deferred*) – A `Deferred` that is expected to fail.
- **exc_types** – The exception types that the Deferred is expected to fail with.
- **failureException** (*type*) – An optional keyword argument. If provided, will raise that exception instead of `testtools.TestCase.failureException`.

Returns A `Deferred` that will fail with an `AssertionError` if `d` does not fail with one of the exception types.

`testtools.twistedsupport.`**flush_logged_errors**(**error_types*)

Flush errors of the given types from the global Twisted log.

Any errors logged during a test will be bubbled up to the test result, marking the test as erroring. Use this function to declare that logged errors were expected behavior.

For example:

```
try:
    1/0
except ZeroDivisionError:
    log.err()
# Prevent logged ZeroDivisionError from failing the test.
flush_logged_errors(ZeroDivisionError)
```

Parameters **error_types** – A variable argument list of exception types.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index

Symbols

A

B

C

D

E

T

U

W