testfixtures Documentation

Release 5.1.1

Simplistix Ltd

Jun 08, 2017

Contents

1 Comparing objects and sequences 3

2 Mocking out objects and methods 17

3 Mocking dates and times 25

4 Testing logging 33

5 Testing output to streams 41

6 Testing with files and directories 43

7 Testing exceptions 55

8 Testing warnings 59

9 Testing use of the subprocess package 61

10 Testing when using django 67

11 Testing with zope.component 69

12 Utilities 71

13 API Reference 75

14 Installation Instructions 89

15 Development 91

16 Changes 93

17 License 107

18 Indices and tables 109

TestFixtures is a collection of helpers and mock objects that are useful when writing unit tests or doc tests.

The sections below describe the use of the various tools included:

CHAPTER 1

Comparing objects and sequences

Python's `unittest` package often fails to give very useful feedback when comparing long sequences or chunks of text. It also has trouble dealing with objects that don't natively support comparison. The functions and classes described here alleviate these problems.

The compare function

The *`compare()`* function can be used as a replacement for `assertEqual()`. It raises an `AssertionError` when its parameters are not equal, which will be reported as a test failure:

```
>>> from testfixtures import compare
>>> compare(1, 2)
Traceback (most recent call last):
  ...
AssertionError: 1 != 2
```

However, it allows you to specify a prefix for the message to be used in the event of failure:

```
>>> compare(1, 2, prefix='wrong number of orders')
Traceback (most recent call last):
  ...
AssertionError: wrong number of orders: 1 != 2
```

This is recommended as it makes the reason for the failure more apparent without having to delve into the code or tests.

You can also optionally specify a suffix, which will be appended to the message on a new line:

```
>>> compare(1, 2, suffix='(Except for very large values of 1)')
Traceback (most recent call last):
  ...
AssertionError: 1 != 2
(Except for very large values of 1)
```

The expected and actual value can also be explicitly supplied, making it clearer as to what has gone wrong:

```
>>> compare(expected=1, actual=2)
Traceback (most recent call last):
 ...
AssertionError: 1 (expected) != 2 (actual)
```

The real strengths of this function come when comparing more complex data types. A number of common python data types will give more detailed output when a comparison fails as described below:

sets

Comparing sets that aren't the same will attempt to highlight where the differences lie:

```
>>> compare(set([1, 2]), set([2, 3]))
Traceback (most recent call last):
 ...
AssertionError: set not as expected:

in first but not second:
[1]

in second but not first:
[3]
```

dicts

Comparing dictionaries that aren't the same will attempt to highlight where the differences lie:

```
>>> compare(dict(x=1, y=2, a=4), dict(x=1, z=3, a=5))
Traceback (most recent call last):
 ...
AssertionError: dict not as expected:

same:
['x']

in first but not second:
'y': 2

in second but not first:
'z': 3

values differ:
'a': 4 != 5
```

lists and tuples

Comparing lists or tuples that aren't the same will attempt to highlight where the differences lie:

```
>>> compare([1, 2, 3], [1, 2, 4])
Traceback (most recent call last):
 ...
```

```
AssertionError: sequence not as expected:

same:
[1, 2]

first:
[3]

second:
[4]
```

namedtuples

When two `namedtuple()` instances are compared, if they are of the same type, the description given will highlight which elements were the same and which were different:

```
>>> from collections import namedtuple
>>> TestTuple = namedtuple('TestTuple', 'x y z')
>>> compare(TestTuple(1, 2, 3), TestTuple(1, 4, 3))
Traceback (most recent call last):
  ...
AssertionError: TestTuple not as expected:

same:
['x', 'z']

values differ:
'y': 2 != 4
```

generators

When two generators are compared, they are both first unwound into tuples and those tuples are then compared.

The *generator* helper is useful for creating a generator to represent the expected results:

```
>>> from testfixtures import generator
>>> def my_gen(t):
...     i = 0
...     while i<t:
...         i += 1
...         yield i
>>> compare(generator(1, 2, 3), my_gen(2))
Traceback (most recent call last):
  ...
AssertionError: sequence not as expected:

same:
(1, 2)

first:
(3,)

second:
()
```

Warning: If you wish to assert that a function returns a generator, say, for performance reasons, then you should use *strict comparison*.

strings and unicodes

Comparison of strings can be tricky, particularly when those strings contain multiple lines; spotting the differences between the expected and actual values can be hard.

To help with this, long strings give a more helpful representation when comparison fails:

```
>>> compare("1234567891011", "1234567789")
Traceback (most recent call last):
  ...
AssertionError:
'1234567891011'
!=
'1234567789'
```

Likewise, multi-line strings give unified diffs when their comparison fails:

```
>>> compare("""
...          This is line 1
...          This is line 2
...          This is line 3
...          """,
...          """
...          This is line 1
...          This is another line
...          This is line 3
...          """)
Traceback (most recent call last):
  ...
AssertionError:
--- first
+++ second
@@ -1,5 +1,5 @@

          This is line 1
-         This is line 2
+         This is another line
          This is line 3
```

Such comparisons can still be confusing as white space is taken into account. If you need to care about whitespace characters, you can make spotting the differences easier as follows:

```
>>> compare("\tline 1\r\nline 2"," line1 \nline 2", show_whitespace=True)
Traceback (most recent call last):
  ...
AssertionError:
--- first
+++ second
@@ -1,2 +1,2 @@
-'\tline 1\r\n'
+' line1 \n'
  'line 2'
```

However, you may not care about some of the whitespace involved. To help with this, *compare()* has two options that can be set to ignore certain types of whitespace.

If you wish to compare two strings that contain blank lines or lines containing only whitespace characters, but where you only care about the content, you can use the following:

```
compare('line1\nline2', 'line1\n \nline2\n\n',
        blanklines=False)
```

If you wish to compare two strings made up of lines that may have trailing whitespace that you don't care about, you can do so with the following:

```
compare('line1\nline2', 'line1 \t\nline2   \n',
        trailing_whitespace=False)
```

Recursive comparison

Where *compare()* is able to provide a descriptive comparison for a particular type, it will then recurse to do the same for the elements contained within objects of that type. For example, when comparing a list of dictionaries, the description will not only tell you where in the list the difference occurred, but also what the differences were within the dictionaries that weren't equal:

```
>>> compare([{'one': 1}, {'two': 2, 'text':'foo\nbar\nbaz'}],
...         [{'one': 1}, {'two': 2, 'text':'foo\nbob\nbaz'}])
Traceback (most recent call last):
 ...
AssertionError: sequence not as expected:

same:
[{'one': 1}]

first:
[{'text': 'foo\nbar\nbaz', 'two': 2}]

second:
[{'text': 'foo\nbob\nbaz', 'two': 2}]

While comparing [1]: dict not as expected:

same:
['two']

values differ:
'text': 'foo\nbar\nbaz' != 'foo\nbob\nbaz'

While comparing [1]['text']:
--- first
+++ second
@@ -1,3 +1,3 @@
 foo
-bar
+bob
 baz
```

This also applies to any comparers you have provided, as can be seen in the next section.

Providing your own comparers

When using *compare()* frequently for your own complex objects, it can be beneficial to give more descriptive output when two objects don't compare as equal.

Note: If you are reading this section as a result of needing to test objects that don't natively support comparison, or as a result of needing to infrequently compare your own subclasses of python basic types, take a look at *Comparison objects* as this may well be an easier solution.

As an example, suppose you have a class whose instances have a timestamp and a name as attributes, but you'd like to ignore the timestamp when comparing:

```
from datetime import datetime

class MyObject(object):
    def __init__(self, name):
        self.timestamp = datetime.now()
        self.name = name
```

To compare lots of these, you would first write a comparer:

```
def compare_my_object(x, y, context):
    if x.name == y.name:
        return
    return 'MyObject named %s != MyObject named %s' % (
        context.label('x', repr(x.name)),
        context.label('y', repr(y.name)),
        )
```

Then you'd register that comparer for your type:

```
from testfixtures.comparison import register
register(MyObject, compare_my_object)
```

Now, it'll get used when comparing objects of that type, even if they're contained within other objects:

```
>>> compare([1, MyObject('foo')], [1, MyObject('bar')])
Traceback (most recent call last):
  ...
AssertionError: sequence not as expected:

same:
[1]

first:
[<MyObject ...>]

second:
[<MyObject ...>]

While comparing [1]: MyObject named 'foo' != MyObject named 'bar'
```

From this example, you can also see that a comparer can indicate that two objects are equal, for *compare()*'s purposes, by returning `None`:

```
>>> MyObject('foo') == MyObject('foo')
False
>>> compare(MyObject('foo'), MyObject('foo'))
```

You can also see that you can, and should, use the context object passed in to add labels to the representations of the objects being compared if the comparison fails:

```
>>> compare(expected=MyObject('foo'), actual=MyObject('bar'))
Traceback (most recent call last):
  ...
AssertionError: MyObject named 'foo' (expected) != MyObject named 'bar' (actual)
```

It may be that you only want to use a comparer or set of comparers for a particular test. If that's the case, you can pass *compare()* a `comparers` parameter consisting of a dictionary that maps types to comparers. These will be added to the global registry for the duration of the call:

```
>>> compare(MyObject('foo'), MyObject('bar'),
...          comparers={MyObject: compare_my_object})
Traceback (most recent call last):
  ...
AssertionError: MyObject named 'foo' != MyObject named 'bar'
```

A full list of the available comparers included can be found below the API documentation for *compare()*. These make good candidates for registering for your own classes, if they provide the necessary behaviour, and their source is also good to read when wondering how to implement your own comparers.

You may be wondering what the `context` object passed to the comparer is for; it allows you to hand off comparison of parts of the two objects currently being compared back to the *compare()* machinery, it also allows you to pass options to your comparison function.

For example, you may have an object that has a couple of dictionaries as attributes:

```
from datetime import datetime

class Request(object):
    def __init__(self, uri, headers, body):
        self.uri = uri
        self.headers = headers
        self.body = body
```

When your tests encounter instances of these that are not as expected, you want feedback about which bits of the request or response weren't as expected. This can be achieved by implementing a comparer as follows:

```
def compare_request(x, y, context):
    uri_different = x.uri != y.uri
    headers_different = context.different(x.headers, y.headers, '.headers')
    body_different = context.different(x.body, y.body, '.body')
    if uri_different or headers_different or body_different:
        return 'Request for %r != Request for %r' % (
            x.uri, y.uri
        )
```

Note: A comparer should always return some text when it considers the two objects it is comparing to be different.

This comparer can either be registered globally or passed to each *compare()* call and will give detailed feedback about how the requests were different:

```
>>> compare(Request('/foo', {'method': 'POST'}, {'my_field': 'value_1'}),
...         Request('/foo', {'method': 'GET'}, {'my_field': 'value_2'}),
...         comparers={Request: compare_request})
Traceback (most recent call last):
 ...
AssertionError: Request for '/foo' != Request for '/foo'

While comparing .headers: dict not as expected:

values differ:
'method': 'POST' != 'GET'

While comparing .headers['method']: 'POST' != 'GET'

While comparing .body: dict not as expected:

values differ:
'my_field': 'value_1' != 'value_2'

While comparing .body['my_field']: 'value_1' != 'value_2'
```

As an example of passing options through to a comparer, suppose you wanted to compare all decimals in a nested data structure by rounding them to a number of decimal places that varies from test to test. The comparer could be implemented and registered as follows:

```
from decimal import Decimal
from testfixtures.comparison import register

def compare_decimal(x, y, context):
     precision = context.get_option('precision', 2)
     if round(x, precision) != round(y, precision):
         return '%r != %r when rounded to %i decimal places' % (
             x, y, precision
         )

register(Decimal, compare_decimal)
```

Now, this comparer will be used for comparing all decimals and the precision used will be that passed to *compare()*:

```
>>> expected_order = {'price': Decimal('1.234'), 'quantity': 5}
>>> actual_order = {'price': Decimal('1.236'), 'quantity': 5}
>>> compare(expected_order, actual_order, precision=1)
>>> compare(expected_order, actual_order, precision=3)
Traceback (most recent call last):
 ...
AssertionError: dict not as expected:

same:
['quantity']

values differ:
'price': Decimal('1.234') != Decimal('1.236')

While comparing ['price']: Decimal('1.234') != Decimal('1.236') when rounded to 3␣
↪decimal places
```

If no precision is passed, the default of 2 will be used:

```
>>> compare(Decimal('2.006'), Decimal('2.009'))
>>> compare(Decimal('2.001'), Decimal('2.009'))
Traceback (most recent call last):
  ...
AssertionError: Decimal('2.001') != Decimal('2.009') when rounded to 2 decimal places
```

Ignoring __eq__

Some objects, such as those from the Django ORM, have pretty broken implementations or `__eq__`. Since *compare()* normally relies on this, it can result in objects appearing to be equal when they are not.

Take this class, for example:

```
class OrmObj(object):
    def __init__(self, a):
        self.a = a
    def __eq__(self, other):
        return True
    def __repr__(self):
        return 'OrmObj: '+str(self.a)
```

If we compare normally, we erroneously understand the objects to be equal:

```
>>> compare(actual=OrmObj(1), expected=OrmObj(2))
```

In order to get a sane comparison, we need to both supply a custom comparer as described above, and use the `ignore_eq` parameter:

```
def compare_orm_obj(x, y, context):
    if x.a != y.a:
        return 'OrmObj: %s != %s' % (x.a, y.a)
```

```
>>> compare(actual=OrmObj(1), expected=OrmObj(2),
...         comparers={OrmObj: compare_orm_obj}, ignore_eq=True)
Traceback (most recent call last):
...
AssertionError: OrmObj: 2 != 1
```

Strict comparison

If is it important that the two values being compared are of exactly the same type, rather than just being equal as far as Python is concerned, then the strict mode of *compare()* should be used.

For example, these two instances will normally appear to be equal provided the elements within them are the same:

```
>>> TypeA = namedtuple('A', 'x')
>>> TypeB = namedtuple('B', 'x')
>>> compare(TypeA(1), TypeB(1))
```

If this type difference is important, then the *strict* parameter should be used:

```
>>> compare(TypeA(1), TypeB(1), strict=True)
Traceback (most recent call last):
  ...
AssertionError: A(x=1) (<class '__main__.A'>) != B(x=1) (<class '__main__.B'>)
```

Comparison objects

Another common problem with the checking in tests is that not all objects support comparison and nor should they need to. For this reason, TextFixtures provides the *Comparison* class.

This class lets you instantiate placeholders that can be used to compare expected results with actual results where objects in the actual results do not support useful comparison.

Comparisons will appear to be equal to any object they are compared with that matches their specification. For example, take the following class:

```
class SomeClass:

    def __init__(self, x, y):
        self.x, self.y = x, y
```

Normal comparison doesn't work, which makes testing tricky:

```
>>> SomeClass(1, 2) == SomeClass(1, 2)
False
```

Here's how this comparison can be done:

```
>>> from testfixtures import Comparison as C
>>> C(SomeClass, x=1, y=2) == SomeClass(1, 2)
True
```

Perhaps even more importantly, when a comparison fails, its representation changes to give information about what went wrong. The common idiom for using comparisons is in conjuction with `assertEqual()` or *compare()*:

```
>>> compare(C(SomeClass, x=2), SomeClass(1, 2))
Traceback (most recent call last):
  ...
AssertionError:
  <C(failed):...SomeClass>
  x:2 != 1
  y:2 not in Comparison
  </C> != <...SomeClass...>
```

The key is that the comparison object actually stores information about what it was last compared with. The following example shows this more clearly:

```
>>> c = C(SomeClass, x=2)
>>> print(repr(c))

  <C:...SomeClass>
  x:2
  </C>
>>> c == SomeClass(1, 2)
False
>>> print(repr(c))

  <C(failed):...SomeClass>
  x:2 != 1
  y:2 not in Comparison
  </C>
```

Note: `assertEqual()` has regressed in Python 3.4 and will now truncate the text shown in assertions with no way to configure this behaviour. Use *`compare()`* instead, which will give you other desirable behaviour as well as showing you the full output of failed comparisons.

Types of comparison

There are several ways a comparison can be set up depending on what you want to check.

If you only care about the class of an object, you can set up the comparison with only the class:

```
>>> C(SomeClass) == SomeClass(1, 2)
True
```

This can also be achieved by specifying the type of the object as a dotted name:

```
>>> import sys
>>> C('types.ModuleType') == sys
True
```

Alternatively, if you happen to have a non-comparable object already around, comparison can be done with it:

```
>>> C(SomeClass(1,2)) == SomeClass(1,2)
True
```

If you only care about certain attributes, this can also easily be achieved with the *strict* parameter:

```
>>> C(SomeClass, x=1, strict=False) == SomeClass(1, 2)
True
```

The above can be problematic if you want to compare an object with attibutes that share names with parameters to the *`Comparison`* constructor. For this reason, you can pass the attributes in a dictionary:

```
>>> compare(C(SomeClass, {'strict':3}, strict=False), SomeClass(1, 2))
Traceback (most recent call last):
  ...
AssertionError:
  <C(failed):...SomeClass>
  strict:3 not in other
  </C> != <...SomeClass...>
```

Gotchas

There are a few things to be careful of when using comparisons:

- The default strict comparison cannot be used with a class such as the following:

  ```
  class NoVars(object):
       __slots__ = ['x']
  ```

  If you try, you will get an error that explains the problem:

  ```
  >>> C(NoVars, x=1) == NoVars()
  Traceback (most recent call last):
  ```

```
 ...
TypeError: <NoVars object at ...> does not support vars() so cannot do strict␣
↪comparison
```

Comparisons can still be done with classes that don't support `vars()`, they just need to be non-strict:

```
>>> nv = NoVars()
>>> nv.x = 1
>>> C(NoVars, x=1, strict=False) == nv
True
```

- If the object being compared has an `__eq__` method, such as Django model instances, then the *Comparison* must be the first object in the equality check.

  The following class is an example of this:

```
class SomeModel:
    def __eq__(self,other):
        if isinstance(other,SomeModel):
            return True
        return False
```

  It will not work correctly if used as the second object in the expression:

```
>>> SomeModel()==C(SomeModel)
False
```

  However, if the comparison is correctly placed first, then everything will behave as expected:

```
>>> C(SomeModel)==SomeModel()
True
```

- It probably goes without saying, but comparisons should not be used on both sides of an equality check:

```
>>> C(SomeClass)==C(SomeClass)
False
```

Round Comparison objects

When comparing numerics you often want to be able to compare to a given precision to allow for rounding issues which make precise equality impossible.

For these situations, you can use *RoundComparison* objects wherever you would use floats or Decimals, and they will compare equal to any float or Decimal that matches when both sides are rounded to the specified precision.

Here's an example:

```
from testfixtures import compare, RoundComparison as R

compare(R(1234.5678, 2), 1234.5681)
```

Note: You should always pass the same type of object to the *RoundComparison* object as you intend compare it with. If the type of the rounded expected value is not the same as the type of the rounded value being compared against it, a `TypeError` will be raised.

Range Comparison objects

When comparing orderable types just as numbers, dates and time, you may only know what range a value will fall into. *RangeComparison* objects let you confirm a value is within a certain tolerance or range.

Here's an example:

```
from testfixtures import compare, RangeComparison as R

compare(R(123.456, 789), Decimal(555.01))
```

Note: *RangeComparison* is inclusive of both the lower and upper bound.

String Comparison objects

When comparing sequences of strings, particularly those comping from things like the python logging package, you often end up wanting to express a requirement that one string should be almost like another, or maybe fit a particular regular expression.

For these situations, you can use *StringComparison* objects wherever you would use normal strings, and they will compare equal to any string that matches the regular expression they are created with.

Here's an example:

```
from testfixtures import compare, StringComparison as S

compare(S('Starting thread \d+'),'Starting thread 132356')
```

Differentiating chunks of text

TextFixtures provides a function that will compare two strings and give a unified diff as a result. This can be handy as a third parameter to `assertEqual()` or just as a general utility function for comparing two lumps of text.

As an example:

```
>>> from testfixtures import diff
>>> print(diff('line1\nline2\nline3',
...            'line1\nlineA\nline3'))
--- first
+++ second
@@ -1,3 +1,3 @@
 line1
-line2
+lineA
 line3
```

CHAPTER 2

Mocking out objects and methods

Mocking is the process of replacing chunks of complex functionality that aren't the subject of the test with mock objects that allow you to check that the mocked out functionality is being used as expected.

In this way, you can break down testing of a complicated set of interacting components into testing of each individual component. The behaviour of components can then be tested individually, irrespective of the behaviour of the components around it.

There are a few implementations of mock objects in the python world. An excellent example and the one recommended for use with TestFixtures is the Mock package: http://pypi.python.org/pypi/mock/

Methods of replacement

TestFixtures provides three different methods of mocking out functionality that can be used to replace functions, classes or even individual methods on a class. Consider the following module:

testfixtures.tests.sample1

```
class X:

    def y(self):
        return "original y"

    @classmethod
    def aMethod(cls):
        return cls

    @staticmethod
    def bMethod():
        return 2
```

We want to mock out the `y` method of the `X` class, with, for example, the following function:

```
def mock_y(self):
     return 'mock y'
```

The context manager

For replacement of a single thing, it's easiest to use the *Replace* context manager:

```
from testfixtures import Replace

def test_function():
    with Replace('testfixtures.tests.sample1.X.y', mock_y):
        print(X().y())
```

For the duration of the `with` block, the replacement is used:

```
>>> test_function()
mock y
```

For multiple replacements to do, or where the you need access to the replacement within the code block under test, the *Replacer* context manager can be used instead:

```
from mock import Mock
from testfixtures import Replacer

def test_function():
    with Replacer() as replace:
        mock_y = replace('testfixtures.tests.sample1.X.y', Mock())
        mock_y.return_value = 'mock y'
        print(X().y())
```

For the duration of the `with` block, the replacement is used:

```
>>> test_function()
mock y
```

The decorator

If you are working in a traditional `unittest` environment and want to replace different things in different test functions, you may find the decorator suits your needs better:

```
from testfixtures import replace

@replace('testfixtures.tests.sample1.X.y', mock_y)
def test_function():
    print(X().y())
```

When using the decorator, the replacement is used for the duration of the decorated callable's execution:

```
>>> test_function()
mock y
```

If you need to manipulate or inspect the object that's used as a replacement, you can add an extra parameter to your function. The decorator will see this and pass the replacement in it's place:

```
from mock import Mock, call
from testfixtures import compare,replace

@replace('testfixtures.tests.sample1.X.y', Mock())
def test_function(mock_y):
    mock_y.return_value = 'mock y'
    print(X().y())
    compare(mock_y.mock_calls, expected=[call()])
```

The above still results in the same output:

```
>>> test_function()
mock y
```

Manual usage

If you want to replace something for the duration of a doctest or you want to replace something for every test in a `TestCase`, then you can use the *`Replacer`* manually.

The instantiation and replacement are done in the `setUp` function of the `TestCase` or passed to the `DocTestSuite` constructor:

```
>>> from testfixtures import Replacer
>>> replace = Replacer()
>>> replace('testfixtures.tests.sample1.X.y', mock_y)
<...>
```

The replacement then stays in place until removed:

```
>>> X().y()
'mock y'
```

Then, in the `tearDown` function of the `TestCase` or passed to the `DocTestSuite` constructor, the replacement is removed:

```
>>> replace.restore()
>>> X().y()
'original y'
```

The *`restore()`* method can also be added as an `addCleanup()` if that is easier or more compact in your test suite.

Replacing more than one thing

Both the *`Replacer`* and the *`replace()`* decorator can be used to replace more than one thing at a time. For the former, this is fairly obvious:

```
def test_function():
    with Replacer() as replace:
        y = replace('testfixtures.tests.sample1.X.y', Mock())
        y.return_value = 'mock y'
        aMethod = replace('testfixtures.tests.sample1.X.aMethod', Mock())
        aMethod.return_value = 'mock method'
```

```
        x = X()
        print(x.y(), x.aMethod())
```

For the decorator, it's less obvious but still pretty easy:

```
from testfixtures import replace

@replace('testfixtures.tests.sample1.X.y', Mock())
@replace('testfixtures.tests.sample1.X.aMethod', Mock())
def test_function(aMethod, y):
    print(aMethod, y)
    aMethod().return_value = 'mock method'
    y().return_value = 'mock y'
    x = X()
    print(aMethod, y)
    print(x.y(), x.aMethod())
```

You'll notice that you can still get access to the replacements, even though there are several of them.

Replacing things that may not be there

The following code shows a situation where `hpy` may or may not be present depending on whether the `guppy` package is installed or not.

testfixtures.tests.sample2

```
try:
    from guppy import hpy
    guppy = True
except ImportError:
    guppy = False


def dump(path):
    if guppy:
        hpy().heap().stat.dump(path)
```

To test the behaviour of the code that uses `hpy` in both of these cases, regardless of whether or not the `guppy` package is actually installed, we need to be able to mock out both `hpy` and the `guppy` global. This is done by doing non-strict replacement, as shown in the following `TestCase`:

```
from testfixtures.tests.sample2 import dump
from testfixtures import replace
from mock import Mock, call

class Tests(unittest.TestCase):

    @replace('testfixtures.tests.sample2.guppy', True)
    @replace('testfixtures.tests.sample2.hpy', Mock(), strict=False)
    def test_method(self, hpy):

        dump('somepath')
```

```
        compare([
                 call(),
                 call().heap(),
                 call().heap().stat.dump('somepath')
                ], hpy.mock_calls)

    @replace('testfixtures.tests.sample2.guppy', False)
    @replace('testfixtures.tests.sample2.hpy', Mock(), strict=False)
    def test_method_no_heapy(self,hpy):

        dump('somepath')

        compare(hpy.mock_calls,[])
```

The *replace()* method and calling a Replacer also supports non-strict replacement using the same keyword parameter.

Replacing items in dictionaries and lists

Replace, *Replacer* and the *replace()* decorator can be used to replace items in dictionaries and lists.

For example, suppose you have a data structure like the following:

testfixtures.tests.sample1

```
someDict = dict(
    key='value',
    complex_key=[1, 2, 3],
    )
```

You can mock out the value associated with key and the second element in the complex_key list as follows:

```
from pprint import pprint
from testfixtures import Replacer
from testfixtures.tests.sample1 import someDict

def test_function():
    with Replacer() as replace:
        replace('testfixtures.tests.sample1.someDict.key', 'foo')
        replace('testfixtures.tests.sample1.someDict.complex_key.1', 42)
        pprint(someDict)
```

While the replacement is in effect, the new items are in place:

```
>>> test_function()
{'complex_key': [1, 42, 3], 'key': 'foo'}
```

When it is no longer in effect, the originals are returned:

```
>>> pprint(someDict)
{'complex_key': [1, 2, 3], 'key': 'value'}
```

Removing attributes and dictionary items

Replace, *Replacer* and the *replace()* decorator can be used to remove attributes from objects and remove items from dictionaries.

For example, suppose you have a data structure like the following:

testfixtures.tests.sample1

```
someDict = dict(
    key='value',
    complex_key=[1, 2, 3],
    )
```

If you want to remove the `key` for the duration of a test, you can do so as follows:

```
from testfixtures import Replace, not_there
from testfixtures.tests.sample1 import someDict

def test_function():
    with Replace('testfixtures.tests.sample1.someDict.key', not_there):
        pprint(someDict)
```

While the replacement is in effect, `key` is gone:

```
>>> test_function()
{'complex_key': [1, 2, 3]}
```

When it is no longer in effect, `key` is returned:

```
>>> pprint(someDict)
{'complex_key': [1, 2, 3], 'key': 'value'}
```

If you want the whole `someDict` dictionary to be removed for the duration of a test, you would do so as follows:

```
from testfixtures import Replace, not_there
from testfixtures.tests import sample1

def test_function():
    with Replace('testfixtures.tests.sample1.someDict', not_there):
        print(hasattr(sample1, 'someDict'))
```

While the replacement is in effect, `key` is gone:

```
>>> test_function()
False
```

When it is no longer in effect, `key` is returned:

```
>>> pprint(sample1.someDict)
{'complex_key': [1, 2, 3], 'key': 'value'}
```

Gotchas

- Make sure you replace the object where it's used and not where it's defined. For example, with the following code from the `testfixtures.tests.sample1` package:

```
from time import time


def str_time():
    return str(time())
```

  You might be tempted to mock things as follows:

```
>>> replace = Replacer()
>>> replace('time.time', Mock())
<...>
```

  But this won't work:

```
>>> from testfixtures.tests.sample1 import str_time
>>> type(float(str_time()))
<... 'float'>
```

  You need to replace `time()` where it's used, not where it's defined:

```
>>> replace('testfixtures.tests.sample1.time', Mock())
<...>
>>> str_time()
"<...Mock...>"
```

  A corollary of this is that you need to replace *all* occurrences of an original to safely be able to test. This can be tricky when an original is imported into many modules that may be used by a particular test.

- You can't replace whole top level modules, and nor should you want to! The reason being that everything up to the last dot in the replacement target specifies where the replacement will take place, and the part after the last dot is used as the name of the thing to be replaced:

```
>>> Replacer().replace('sys', Mock())
Traceback (most recent call last):
  ...
ValueError: target must contain at least one dot!
```

CHAPTER 3

Mocking dates and times

Testing code that involves dates and times or which has behaviour dependent on the date or time it is executed at has historically been tricky. Mocking lets you perform tests on this type of code and TestFixtures provides three specialised mock objects to help with this.

Dates

TestFixtures provides the *test_date()* function that returns a subclass of `datetime.date` with a `today()` method that will return a consistent sequence of dates each time it is called.

This enables you to write tests for code such as the following, from the `testfixtures.tests.sample1` package:

```
from datetime import datetime, date


def str_today_1():
    return str(date.today())
```

Replace can be used to apply the mock as shown in the following example, which could appear in either a unit test or a doc test:

```
>>> from testfixtures import Replace, test_date
>>> from testfixtures.tests.sample1 import str_today_1
>>> with Replace('testfixtures.tests.sample1.date', test_date()):
...     str_today_1()
...     str_today_1()
'2001-01-01'
'2001-01-02'
```

If you need a specific date to be returned, you can specify it:

```
>>> with Replace('testfixtures.tests.sample1.date', test_date(1978,6,13)):
...     str_today_1()
'1978-06-13'
```

If you need to test with a whole sequence of specific dates, this can be done as follows:

```
>>> with Replace('testfixtures.tests.sample1.date', test_date(None)) as d:
...     d.add(1978,6,13)
...     d.add(2009,11,12)
...     str_today_1()
...     str_today_1()
'1978-06-13'
'2009-11-12'
```

Another way to test with a specific sequence of dates is to use the `delta_type` and `delta` parameters to *`test_date()`*. These parameters control the type and size, respectively, of the difference between each date returned.

For example, where 2 days elapse between each returned value:

```
>>> with Replace('testfixtures.tests.sample1.date',
...               test_date(1978, 6, 13, delta=2, delta_type='days')) as d:
...     str_today_1()
...     str_today_1()
...     str_today_1()
'1978-06-13'
'1978-06-15'
'1978-06-17'
```

The `delta_type` can be any keyword parameter accepted by the `timedelta` constructor. Specifying a `delta` of zero can be an effective way of ensuring that all calls to the `today()` method return the same value:

```
>>> with Replace('testfixtures.tests.sample1.date',
...               test_date(1978, 6, 13, delta=0)) as d:
...     str_today_1()
...     str_today_1()
...     str_today_1()
'1978-06-13'
'1978-06-13'
'1978-06-13'
```

When using *`test_date()`*, you can, at any time, set the next date to be returned using the `set()` method. The date returned after this will be the set date plus the `delta` in effect:

```
>>> with Replace('testfixtures.tests.sample1.date', test_date(delta=2)) as d:
...     str_today_1()
...     d.set(1978,8,1)
...     str_today_1()
...     str_today_1()
'2001-01-01'
'1978-08-01'
'1978-08-03'
```

Datetimes

TextFixtures provides the *test_datetime()* function that returns a subclass of datetime.datetime with a now() method that will return a consistent sequence of datetime objects each time it is called.

This enables you to write tests for code such as the following, from the testfixtures.tests.sample1 package:

```
from datetime import datetime, date


def str_now_1():
    return str(datetime.now())
```

We use the a *Replacer* as follows, which could appear in either a unit test or a doc test:

```
>>> from testfixtures import Replacer, test_datetime
>>> from testfixtures.tests.sample1 import str_now_1
>>> with Replace('testfixtures.tests.sample1.datetime', test_datetime()):
...     str_now_1()
...     str_now_1()
'2001-01-01 00:00:00'
'2001-01-01 00:00:10'
```

If you need a specific datetime to be returned, you can specify it:

```
>>> with Replace('testfixtures.tests.sample1.datetime',
...               test_datetime(1978,6,13,1,2,3)):
...     str_now_1()
'1978-06-13 01:02:03'
```

If you need to test with a whole sequence of specific datetimes, this can be done as follows:

```
>>> with Replace('testfixtures.tests.sample1.datetime',
...               test_datetime(None)) as d:
...     d.add(1978,6,13,16,0,1)
...     d.add(2009,11,12,11,41,20)
...     str_now_1()
...     str_now_1()
'1978-06-13 16:00:01'
'2009-11-12 11:41:20'
```

Another way to test with a specific sequence of datetimes is to use the delta_type and delta parameters to *test_datetime()*. These parameters control the type and size, respectively, of the difference between each datetime returned.

For example, where 2 hours elapse between each returned value:

```
>>> with Replace(
...     'testfixtures.tests.sample1.datetime',
...     test_datetime(1978, 6, 13, 16, 0, 1, delta=2, delta_type='hours')
... ) as d:
...     str_now_1()
...     str_now_1()
...     str_now_1()
'1978-06-13 16:00:01'
'1978-06-13 18:00:01'
'1978-06-13 20:00:01'
```

The `delta_type` can be any keyword parameter accepted by the `timedelta` constructor. Specifying a `delta` of zero can be an effective way of ensuring that all calls to the `now()` method return the same value:

```
>>> with Replace('testfixtures.tests.sample1.datetime',
...               test_datetime(1978, 6, 13, 16, 0, 1, delta=0)) as d:
...     str_now_1()
...     str_now_1()
...     str_now_1()
'1978-06-13 16:00:01'
'1978-06-13 16:00:01'
'1978-06-13 16:00:01'
```

When using *`test_datetime()`*, you can, at any time, set the next datetime to be returned using the `set()` method. The value returned after this will be the set value plus the `delta` in effect:

```
>>> with Replace('testfixtures.tests.sample1.datetime',
...               test_datetime(delta=2)) as d:
...     str_now_1()
...     d.set(1978,8,1)
...     str_now_1()
...     str_now_1()
'2001-01-01 00:00:00'
'1978-08-01 00:00:00'
'1978-08-01 00:00:02'
```

Timezones

In many situations where you're mocking out `now()` or `utcnow()` you're not concerned about timezones, especially given that both methods will usually return `datetime` objects that have a *tzinfo* of `None`. However, in some applications it is important that `now()` and `utcnow()` return different times, as they would normally if the application is run anywhere other than the UTC timezone.

The best way to understand how to use *`test_datetime()`* in these situations is to think of the internal queue as being a queue of `datetime` objects at the current local time with a *tzinfo* of None, much as would be returned by `now()`. If you pass in a *tz* parameter to *`now()`* it will be applied to the value before it is returned in the same way as it would by `datetime.datetime.now()`.

If you pass in a *tzinfo* to *`test_datetime()`*, this will be taken to indicate the timezone you intend for the local times that *`now()`* simulates. As such, that timezone will be used to compute values returned from *`utcnow()`* such that they would be *`test_datetime`* objects in the UTC timezone with the *tzinfo* set to `None`, as would be the case for a normal call to `datetime.datetime.utcnow()`.

For example, lets take a timezone as defined by the following class:

```
from datetime import tzinfo, timedelta

class ATZInfo(tzinfo):

   def tzname(self, dt):
        return 'A TimeZone'

   def utcoffset(self, dt):
        # In general, this timezone is 5 hours behind UTC
        offset  = timedelta(hours=-5)
        return offset+self.dst(dt)

   def dst(self, dt):
```

```
        # However, between March and September, it is only
        # 4 hours behind UTC
        if 3 < dt.month < 9:
            return timedelta(hours=1)
        return timedelta()
```

If we create a *test_datetime* with this timezone and a delta of zero, so we can see affect of the timezone over multiple calls, the values returned by *now()* will be affected:

```
>>> datetime = test_datetime(2001, 1, 1, delta=0, tzinfo=ATZInfo())
```

A normal call to *now()* will return the values passed to the constructor:

```
>>> print(datetime.now())
2001-01-01 00:00:00
```

If we now ask for this time but in the timezone we passed to *test_datetime*, we will get the same hours, minutes and seconds but with a `tzinfo` attribute set:

```
>>> print(datetime.now(ATZInfo()))
2001-01-01 00:00:00-05:00
```

If we call *utcnow()*, we will get the time equivalent to the values passed to the constructor, but in the UTC timezone:

```
>>> print(datetime.utcnow())
2001-01-01 05:00:00
```

The timezone passed in when the *test_datetime* is created has a similar effect on any items set:

```
>>> datetime.set(2011,5,1,10)
>>> print(datetime.now())
2011-05-01 10:00:00
>>> print(datetime.utcnow())
2011-05-01 14:00:00
```

Likewise, *add()* behaves the same way:

```
>>> datetime = test_datetime(None, delta=0, tzinfo=ATZInfo())
>>> datetime.add(2011,1,1,10)
>>> datetime.add(2011,5,1,10)
>>> datetime.add(2011,10,1,10)
>>> print(datetime.now())
2011-01-01 10:00:00
>>> print(datetime.utcnow())
2011-05-01 14:00:00
>>> print(datetime.now())
2011-10-01 10:00:00
```

Times

TextFixtures provides the *test_time()* function that, when called, returns a replacement for the `time.time()` function.

This enables you to write tests for code such as the following, from the `testfixtures.tests.sample1` package:

```
from time import time


def str_time():
    return str(time())
```

We use the a *Replacer* as follows, which could appear in either a unit test or a doc test:

```
>>> from testfixtures import Replacer, test_time
>>> from testfixtures.tests.sample1 import str_time
>>> with Replace('testfixtures.tests.sample1.time', test_time()):
...     str_time()
...     str_time()
'978307200.0'
'978307201.0'
```

If you need an integer representing a specific time to be returned, you can specify it:

```
>>> with Replace('testfixtures.tests.sample1.time',
...              test_time(1978, 6, 13, 1, 2, 3)):
...     str_time()
'266547723.0'
```

If you need to test with a whole sequence of specific timestamps, this can be done as follows:

```
>>> with Replace('testfixtures.tests.sample1.time', test_time(None)) as t:
...     t.add(1978,6,13,16,0,1)
...     t.add(2009,11,12,11,41,20)
...     str_time()
...     str_time()
'266601601.0'
'1258026080.0'
```

Another way to test with a specific sequence of timestamps is to use the `delta_type` and `delta` parameters to *test_time()*. These parameters control the type and size, respectively, of the difference between each timestamp returned.

For example, where 2 hours elapse between each returned value:

```
>>> with Replace(
...      'testfixtures.tests.sample1.time',
...      test_time(1978, 6, 13, 16, 0, 1, delta=2, delta_type='hours')
... ) as d:
...     str_time()
...     str_time()
...     str_time()
'266601601.0'
'266608801.0'
'266616001.0'
```

The `delta_type` can be any keyword parameter accepted by the `timedelta` constructor. Specifying a `delta` of zero can be an effective way of ensuring that all calls to the `time()` function return the same value:

```
>>> with Replace('testfixtures.tests.sample1.time',
...              test_time(1978, 6, 13, 16, 0, 1, delta=0)) as d:
...     str_time()
...     str_time()
...     str_time()
```

```
'266601601.0'
'266601601.0'
'266601601.0'
```

When using *test_time()*, you can, at any time, set the next timestamp to be returned using the `set()` method. The value returned after this will be the set value plus the `delta` in effect:

```
>>> with Replace('testfixtures.tests.sample1.time', test_time(delta=2)) as d:
...     str_time()
...     d.set(1978,8,1)
...     str_time()
...     str_time()
'978307200.0'
'270777600.0'
'270777602.0'
```

Gotchas with dates and times

Using these specialised mock objects can have some intricacies as described below:

Local references to functions

There are situations where people may have obtained a local reference to the `today()` or `now()` methods, such as the following code from the `testfixtures.tests.sample1` package:

```
from datetime import datetime, date


now = datetime.now


def str_now_2():
    return str(now())
today = date.today


def str_today_2():
    return str(today())
```

In these cases, you need to be careful with the replacement:

```
>>> from testfixtures import Replacer, test_datetime
>>> from testfixtures.tests.sample1 import str_now_2, str_today_2
>>> with Replacer() as replace:
...     today = replace('testfixtures.tests.sample1.today', test_date().today)
...     now = replace('testfixtures.tests.sample1.now', test_datetime().now)
...     str_today_2()
...     str_now_2()
'2001-01-01'
'2001-01-01 00:00:00'
```

Use with code that checks class types

When using the above specialist mocks, you may find code that checks the type of parameters passed may get confused. This is because, by default, *test_datetime* and *test_date* return instances of the real datetime and date classes:

```
>>> from testfixtures import test_datetime
>>> from datetime import datetime
>>> tdatetime = test_datetime()
>>> issubclass(tdatetime, datetime)
True
>>> tdatetime.now().__class__
<...'datetime.datetime'>
```

The above behaviour, however, is generally what you want as other code in your application and, more importantly, in other code such as database adapters, may handle instances of the real datetime and date classes, but not instances of the *test_datetime* and *test_date* mocks.

That said, this behaviour can cause problems if you check the type of an instance against one of the mock classes. Most people might expect the following to return True:

```
>>> isinstance(tdatetime(2011, 1, 1), tdatetime)
False
>>> isinstance(tdatetime.now(), tdatetime)
False
```

If this causes a problem for you, then both datetime and date take a *strict* keyword parameter that can be used as follows:

```
>>> tdatetime = test_datetime(strict=True)
>>> tdatetime.now().__class__
<class 'testfixtures.tdatetime.tdatetime'>
>>> isinstance(tdatetime.now(), tdatetime)
True
```

You will need to take care that you have replaced occurrences of the class where type checking is done with the correct *test_datetime* or *test_date*. Also, be aware that the *date()* method of *test_datetime* instances will still return a normal date instance. If type checking related to this is causing problems, the type the *date()* method returns can be controlled as shown in the following example:

```
from testfixtures import test_date, test_datetime

date_type = test_date(strict=True)
datetime_type = test_datetime(strict=True, date_type=date_type)
```

With things set up like this, the *date()* method will return an instance of the date_type mock:

```
>>> somewhen = datetime_type.now()
>>> somewhen.date()
tdate(2001, 1, 1)
>>> _.__class__ is date_type
True
```

CHAPTER 4

Testing logging

Python includes an excellent `logging` package, however many people assume that logging calls do not need to be tested. They may also want to test logging calls but find the prospect too daunting. To help with this, TestFixtures allows you to easily capture the output of calls to Python's logging framework and make sure they were as expected.

Note: The *`LogCapture`* class is useful for checking that your code logs the right messages. If you want to check that the configuration of your handlers is correct, please see the *section* below.

Methods of capture

There are three different techniques for capturing messages logged to the Python logging framework, depending on the type of test you are writing. They are all described in the sections below.

The context manager

If you're using a version of Python where the `with` keyword is available, the context manager provided by TestFixtures can be used:

```
>>> import logging
>>> from testfixtures import LogCapture
>>> with LogCapture() as l:
...     logger = logging.getLogger()
...     logger.info('a message')
...     logger.error('an error')
```

For the duration of the `with` block, log messages are captured. The context manager provides a check method that raises an exception if the logging wasn't as you expected:

```
>>> l.check(
...     ('root', 'INFO', 'a message'),
```

```
...      ('root', 'ERROR', 'another error'),
...      )
Traceback (most recent call last):
 ...
AssertionError: sequence not as expected:

same:
(('root', 'INFO', 'a message'),)

expected:
(('root', 'ERROR', 'another error'),)

actual:
(('root', 'ERROR', 'an error'),)
```

It also has a string representation that allows you to see what has been logged, which is useful for doc tests:

```
>>> print(l)
root INFO
  a message
root ERROR
  an error
```

The decorator

If you are working in a traditional `unittest` environment and only want to capture logging for a particular test function, you may find the decorator suits your needs better:

```
from testfixtures import log_capture

@log_capture()
def test_function(l):
    logger = logging.getLogger()
    logger.info('a message')
    logger.error('an error')

    l.check(
        ('root', 'INFO', 'a message'),
        ('root', 'ERROR', 'an error'),
        )
```

Manual usage

If you want to capture logging for the duration of a doctest or in every test in a `TestCase`, then you can use the *`LogCapture`* manually.

The instantiation and replacement are done in the `setUp` function of the `TestCase` or passed to the `DocTestSuite` constructor:

```
>>> from testfixtures import LogCapture
>>> l = LogCapture()
```

You can then execute whatever will log the messages you want to test for:

```
>>> from logging import getLogger
>>> getLogger().info('a message')
```

At any point, you can check what has been logged using the check method:

```
>>> l.check(('root', 'INFO', 'a message'))
```

Alternatively, you can use the string representation of the *LogCapture*:

```
>>> print(l)
root INFO
  a message
```

Then, in the `tearDown` function of the `TestCase` or passed to the `DocTestSuite` constructor, you should make sure you stop the capturing:

```
>>> l.uninstall()
```

If you have multiple *LogCapture* objects in use, you can easily uninstall them all:

```
>>> LogCapture.uninstall_all()
```

Checking captured log messages

Regardless of how you use the *LogCapture* to capture messages, there are three ways of checking that the messages captured were as expected.

The following example is useful for showing these:

```
from testfixtures import LogCapture
from logging import getLogger
logger = getLogger()

with LogCapture() as l:
    logger.info('start of block number %i', 1)
    try:
        raise RuntimeError('No code to run!')
    except:
        logger.error('error occurred', exc_info=True)
```

The check method

The *LogCapture* has a *check()* method that will compare the log messages captured with those you expect. Expected messages are expressed as three-element tuples where the first element is the name of the logger to which the message should have been logged, the second element is the string representation of the level at which the message should have been logged and the third element is the message that should have been logged after any parameter interpolation has taken place.

If things are as you expected, the method will not raise any exceptions:

```
>>> result = l.check(
...     ('root', 'INFO', 'start of block number 1'),
...     ('root', 'ERROR', 'error occurred'),
...     )
```

However, if the actual messages logged were different, you'll get an `AssertionError` explaining what happened:

```
>>> l.check(('root', 'INFO', 'start of block number 1'))
Traceback (most recent call last):
  ...
AssertionError: sequence not as expected:

same:
(('root', 'INFO', 'start of block number 1'),)

expected:
()

actual:
(('root', 'ERROR', 'error occurred'),)
```

Printing

The *LogCapture* has a string representation that shows what messages it has captured. This can be useful in doc tests:

```
>>> print(l)
root INFO
  start of block number 1
root ERROR
  error occurred
```

This representation can also be used to check that no logging has occurred:

```
>>> empty = LogCapture()
>>> print(empty)
No logging captured
```

Inspecting

The *LogCapture* also keeps a list of the `LogRecord` instances it captures. This is useful when you want to check specifics of the captured logging that aren't available from either the string representation or the *check()* method.

A common case of this is where you want to check that exception information was logged for certain messages:

```
>>> print(l.records[-1].exc_info)
(<... '...RuntimeError'>, RuntimeError('No code to run!',), <traceback object at ...>)
```

If you're working in a unit test, the following code may be more appropriate:

```
from testfixtures import compare, Comparison as C

compare(C(RuntimeError('No code to run!')), l.records[-1].exc_info[1])
```

Only capturing specific logging

Some actions that you want to test may generate a lot of logging, only some of which you actually need to care about.

The logging you care about is often only that above a certain log level. If this is the case, you can configure *LogCapture* to only capture logging at or above a specific level.

If using the context manager, you would do this:

```
>>> with LogCapture(level=logging.INFO) as l:
...     logger = getLogger()
...     logger.debug('junk')
...     logger.info('something we care about')
...     logger.error('an error')
>>> print(l)
root INFO
  something we care about
root ERROR
  an error
```

If using the decorator, you would do this:

```
@log_capture(level=logging.INFO)
def test_function(l):
    logger= getLogger()
    logger.debug('junk')
    logger.info('what we care about')

    l.check(('root', 'INFO', 'what we care about'))
```

In other cases this problem can be alleviated by only capturing a specific logger.

If using the context manager, you would do this:

```
>>> with LogCapture('specific') as l:
...     getLogger('something').info('junk')
...     getLogger('specific').info('what we care about')
...     getLogger().info('more junk')
>>> print(l)
specific INFO
  what we care about
```

If using the decorator, you would do this:

```
@log_capture('specific')
def test_function(l):
    getLogger('something').info('junk')
    getLogger('specific').info('what we care about')
    getLogger().info('more junk')

    l.check(('specific', 'INFO', 'what we care about'))
```

However, it may be that while you don't want to capture all logging, you do want to capture logging from multiple specific loggers.

You would do this with the context manager as follows:

```
>>> with LogCapture(('one','two')) as l:
...     getLogger('three').info('3')
...     getLogger('two').info('2')
...     getLogger('one').info('1')
>>> print(l)
two INFO
```

```
  2
one INFO
  1
```

Likewise, the same thing can be done with the decorator:

```
@log_capture('one','two')
def test_function(l):
    getLogger('three').info('3')
    getLogger('two').info('2')
    getLogger('one').info('1')

    l.check(
        ('two', 'INFO', '2'),
        ('one', 'INFO', '1')
        )
```

It may also be that the simplest thing to do is only capture logging for part of your test. This is particularly common with long doc tests. To make this easier, *LogCapture* supports manual installation and uninstallation as shown in the following example:

```
>>> l = LogCapture(install=False)
>>> getLogger().info('junk')
>>> l.install()
>>> getLogger().info('something we care about')
>>> l.uninstall()
>>> getLogger().info('more junk')
>>> l.install()
>>> getLogger().info('something else we care about')
>>> print(l)
root INFO
  something we care about
root INFO
  something else we care about
```

Checking the configuration of your log handlers

LogCapture is good for checking that your code is logging the correct messages; just as important is checking that your application has correctly configured log handers. This can be done using a unit test such as the following:

```
from testfixtures import Comparison as C, compare
from unittest import TestCase
import logging
import sys

class LoggingConfigurationTests(TestCase):

    # We mock out the handlers list for the logger we're
    # configuring in such a way that we have no handlers
    # configured at the start of the test and the handlers our
    # configuration installs are removed at the end of the test.

    def setUp(self):
        self.logger = logging.getLogger()
        self.orig_handlers = self.logger.handlers
```

```
        self.logger.handlers = []
        self.level = self.logger.level

    def tearDown(self):
        self.logger.handlers = self.orig_handlers
        self.logger.level = self.level

    def test_basic_configuration(self):
        # Our logging configuration code, in this case just a
        # call to basicConfig:
        logging.basicConfig(format='%(levelname)s %(message)s',
                            level=logging.INFO)

        # Now we check the configuration is as expected:

        compare(self.logger.level, 20)
        compare([
            C('logging.StreamHandler',
              stream=sys.stderr,
              formatter=C('logging.Formatter',
                          _fmt='%(levelname)s %(message)s',
                          strict=False),
              level=logging.NOTSET,
              strict=False)
            ], self.logger.handlers)
```

CHAPTER 5

Testing output to streams

In many situations, it's perfectly legitimate for output to be printed to one of the standard streams. To aid with testing this kind of output, TestFixtures provides the *OutputCapture* helper.

This helper is a context manager that captures output sent to `sys.stdout` and `sys.stderr` and provides a *compare()* method to check that the output was as expected.

Here's a simple example:

```
from testfixtures import OutputCapture
import sys

with OutputCapture() as output:
    # code under test
    print("Hello!")
    print("Something bad happened!", file=sys.stderr)

output.compare('\n'.join([
    "Hello!",
    "Something bad happened!",
]))
```

To make life easier, both the actual and expected output are stripped of leading and trailing whitespace before the comparison is done:

```
>>> with OutputCapture() as o:
...     print('  Bar! ')
...     o.compare(' Foo!  ')
Traceback (most recent call last):
...
AssertionError: 'Foo!' (expected) != 'Bar!' (actual)
```

However, if you need to make very explicit assertions about what has been written to the stream then you can do so using the *captured* property of the *OutputCapture*:

```
>>> with OutputCapture() as o:
...     print('  Bar! ')
>>> print(repr(o.captured))
'  Bar! \n'
```

If you need to explicitly check whether output went to `stdout` or `stderr`, *separate* mode can be used:

```
from testfixtures import OutputCapture
import sys

with OutputCapture(separate=True) as output:
    print("Hello!")
    print("Something bad happened!", file=sys.stderr)

output.compare(
    stdout="Hello!",
    stderr="Something bad happened!",
    )
```

Finally, you may sometimes want to disable an *OutputCapture* without removing it from your code. This often happens when you want to insert a debugger call while an *OutputCapture* is active; if it remains enabled, all debugger output will be captured making the debugger very difficult to use!

To deal with this problem, the *OutputCapture* may be disabled and then re-enabled as follows:

```
>>> with OutputCapture() as o:
...     print('Foo')
...     o.disable()
...     print('Bar')
...     o.enable()
...     print('Baz')
Bar
>>> print(o.captured)
Foo
Baz
```

Note: Some debuggers, notably pdb, do interesting things with streams such that calling *disable()* from within the debugger will have no effect. A good fallback is to type the following, which will almost always restore output to where you want it:

```
import sys; sys.stdout=sys.__stdout__
```

CHAPTER 6

Testing with files and directories

Working with files and directories in tests can often require excessive amounts of boilerplate code to make sure that the tests happen in their own sandbox, files and directories contain what they should or code processes test files correctly, and the sandbox is cleared up at the end of the tests.

Methods of use

To help with this, TestFixtures provides the *TempDirectory* class that hides most of the boilerplate code you would need to write.

Suppose you wanted to test the following function:

```
import os

def foo2bar(dirpath, filename):
  path = os.path.join(dirpath, filename)
  with open(path, 'rb') as input:
      data = input.read()
  data = data.replace(b'foo', b'bar')
  with open(path, 'wb') as output:
      output.write(data)
```

There are several different ways depending on the type of test you are writing:

The context manager

If you're using a version of Python where the `with` keyword is available, a *TempDirectory* can be used as a context manager:

```
>>> from testfixtures import TempDirectory
>>> with TempDirectory() as d:
...   d.write('test.txt', b'some foo thing')
```

```
...    foo2bar(d.path, 'test.txt')
...    d.read('test.txt')
'...'
b'some bar thing'
```

The decorator

If you are working in a traditional `unittest` environment and only work with files or directories in a particular test function, you may find the decorator suits your needs better:

```
from testfixtures import tempdir, compare

@tempdir()
def test_function(d):
    d.write('test.txt', b'some foo thing')
    foo2bar(d.path, 'test.txt')
    compare(d.read('test.txt'), b'some bar thing')
```

Manual usage

If you want to work with files or directories for the duration of a doctest or in every test in a `TestCase`, then you can use the *`TempDirectory`* manually.

The instantiation and replacement are done in the `setUp` function of the `TestCase` or passed to the `DocTestSuite` constructor:

```
>>> from testfixtures import TempDirectory
>>> d = TempDirectory()
```

You can then use the temporary directory for your testing:

```
>>> d.write('test.txt', b'some foo thing')
'...'
>>> foo2bar(d.path, 'test.txt')
>>> d.read('test.txt') == b'some bar thing'
True
```

Then, in the `tearDown` function of the `TestCase` or passed to the `DocTestSuite` constructor, you should make sure the temporary directory is cleaned up:

```
>>> d.cleanup()
```

If you have multiple *`TempDirectory`* objects in use, you can easily clean them all up:

```
>>> TempDirectory.cleanup_all()
```

Features of a temporary directory

No matter which usage pattern you pick, you will always end up with a *`TempDirectory`* object. These have an array of methods that let you perform common file and directory related tasks without all the manual boiler plate. The following sections show you how to perform the various tasks you're likely to bump into in the course of testing.

Computing paths

If you need to know the real path of the temporary directory, the *TempDirectory* object has a *path* attribute:

```
>>> tempdir.path
'...tmp...'
```

A common use case is to want to compute a path within the temporary directory to pass to code under test. This can be done with the *getpath()* method:

```
>>> tempdir.getpath('foo').rsplit(os.sep,1)[-1]
'foo'
```

If you want to compute a deeper path, you can either pass either a tuple or a forward slash-separated path:

```
>>> tempdir.getpath(('foo', 'baz')).rsplit(os.sep, 2)[-2:]
['foo', 'baz']
>>> tempdir.getpath('foo/baz') .rsplit(os.sep, 2)[-2:]
['foo', 'baz']
```

Note: If passing a string containing path separators, a forward slash should be used as the separator regardless of the underlying platform separator.

Writing files

To write to a file in the root of the temporary directory, you pass the name of the file and the content you want to write:

```
>>> tempdir.write('myfile.txt', b'some text')
'...'
>>> with open(os.path.join(tempdir.path, 'myfile.txt')) as f:
...     print(f.read())
some text
```

The full path of the newly written file is returned:

```
>>> path = tempdir.write('anotherfile.txt', b'some more text')
>>> with open(path) as f:
...     print(f.read())
some more text
```

You can also write files into a sub-directory of the temporary directory, whether or not that directory exists, as follows:

```
>>> path = tempdir.write(('some', 'folder', 'afile.txt'), b'the text')
>>> with open(path) as f:
...     print(f.read())
the text
```

You can also specify the path to write to as a forward-slash separated string:

```
>>> path = tempdir.write('some/folder/bfile.txt', b'the text')
>>> with open(path) as f:
...     print(f.read())
the text
```

Note: Forward slashes should be used regardless of the file system or operating system in use.

Creating directories

If you just want to create a sub-directory in the temporary directory you can do so as follows:

```
>>> tempdir.makedir('output')
'...'
>>> os.path.isdir(os.path.join(tempdir.path, 'output'))
True
```

As with file creation, the full path of the sub-directory that has just been created is returned:

```
>>> path = tempdir.makedir('more_output')
>>> os.path.isdir(path)
True
```

Finally, you can create a nested sub-directory even if the intervening parent directories do not exist:

```
>>> os.path.exists(os.path.join(tempdir.path, 'some'))
False
>>> path = tempdir.makedir(('some', 'sub', 'dir'))
>>> os.path.exists(path)
True
```

You can also specify the path to write to as a forward-slash separated string:

```
>>> os.path.exists(os.path.join(tempdir.path, 'another'))
False
>>> path = tempdir.makedir('another/sub/dir')
>>> os.path.exists(path)
True
```

Note: Forward slashes should be used regardless of the file system or operating system in use.

Checking the contents of files

Once a file has been written into the temporary directory, you will often want to check its contents. This is done with the *TempDirectory.read()* method.

Suppose the code you are testing creates some files:

```
def spew(path):
    with open(os.path.join(path, 'root.txt'), 'wb') as f:
        f.write(b'root output')
    os.mkdir(os.path.join(path, 'subdir'))
    with open(os.path.join(path, 'subdir', 'file.txt'), 'wb') as f:
        f.write(b'subdir output')
    os.mkdir(os.path.join(path, 'subdir', 'logs'))
```

We can test this function by passing it the temporary directory's path and then using the *TempDirectory.read()* method to check the files were created with the correct content:

```
>>> spew(tempdir.path)
>>> tempdir.read('root.txt')
b'root output'
>>> tempdir.read(('subdir', 'file.txt'))
b'subdir output'
```

The second part of the above test shows how to use the *TempDirectory.read()* method to check the contents of files that are in sub-directories of the temporary directory. This can also be done by specifying the path relative to the root of the temporary directory as a forward-slash separated string:

```
>>> tempdir.read('subdir/file.txt')
b'subdir output'
```

Note: Forward slashes should be used regardless of the file system or operating system in use.

Checking the contents of directories

It's good practice to test that your code is only writing files you expect it to and to check they are being written to the path you expect. *TempDirectory.compare()* is the method to use to do this.

As an example, we could check that the `spew()` function above created no extraneous files as follows:

```
>>> tempdir.compare([
...     'root.txt',
...     'subdir/',
...     'subdir/file.txt',
...     'subdir/logs/',
... ])
```

If we only wanted to check the sub-directory, we would specify the path to start from, relative to the root of the temporary directory:

```
>>> tempdir.compare([
...     'file.txt',
...     'logs/',
... ], path='subdir')
```

If, like git, we only cared about files, we could do the comparison as follows:

```
>>> tempdir.compare([
...     'root.txt',
...     'subdir/file.txt',
... ], files_only=True)
```

And finally, if we only cared about files at a particular level, we could turn off the recursive comparison as follows:

```
>>> tempdir.compare([
...     'root.txt',
...     'subdir',
... ], recursive=False)
```

The *compare()* method can also be used to check whether a directory contains nothing, for example:

```
>>> tempdir.compare(path=('subdir', 'logs'), expected=())
```

The above can also be done by specifying the sub-directory to be checked as a forward-slash separated path:

```
>>> tempdir.compare(path='subdir/logs', expected=())
```

If the actual directory contents do not match the expected contents passed in, an `AssertionError` is raised, which will show up as a unit test failure:

```
>>> tempdir.compare(['subdir'], recursive=False)
Traceback (most recent call last):
...
AssertionError: sequence not as expected:

same:
()

expected:
('subdir',)

actual:
('root.txt', 'subdir')
```

In some circumstances, you may want to ignore certain files or sub-directories when checking contents. To make this easy, the *TempDirectory* constructor takes an optional *ignore* parameter which, if provided, should contain a sequence of regular expressions. If any of the regular expressions return a match when used to search through the results of any of the the methods covered in this section, that result will be ignored.

For example, suppose we are testing some revision control code, but don't really care about the revision control system's metadata directories, which may or may not be present:

```
from random import choice

def svn_ish(dirpath, filename):
  if choice((True, False)):
    os.mkdir(os.path.join(dirpath, '.svn'))
  with open(os.path.join(dirpath, filename), 'wb') as f:
    f.write(b'something')
```

To test this, we can use any of the previously described methods.

When used manually or as a context manager, this would be as follows:

```
>>> with TempDirectory(ignore=['.svn']) as d:
...   svn_ish(d.path, 'test.txt')
...   d.compare(['test.txt'])
```

The decorator would be as follows:

```
from testfixtures import tempdir, compare

@tempdir(ignore=['.svn'])
def test_function(d):
    svn_ish(d.path, 'test.txt')
    d.compare(['test.txt'])
```

If you are working with doctests, the *listdir()* method can be used instead:

```
>>> tempdir.listdir()
root.txt
subdir
>>> tempdir.listdir('subdir')
file.txt
logs
>>> tempdir.listdir(('subdir', 'logs'))
No files or directories found.
```

The above example also shows how to check the contents of sub-directories of the temporary directory and also shows what is printed when a directory contains nothing. The *listdir()* method can also take a path separated by forward slashes, which can make doctests a little more readable. The above test could be written as follows:

```
>>> tempdir.listdir('subdir/logs')
No files or directories found.
```

However, if you have a nested folder structure, such as that created by our `spew()` function, it can be easier to just inspect the whole tree of files and folders created. You can do this by using the *recursive* parameter to *listdir()*:

```
>>> tempdir.listdir(recursive=True)
root.txt
subdir/
subdir/file.txt
subdir/logs/
```

Bytes versus Strings

You'll notice that all of the examples so far have used raw bytes as their data and written to and read from files only in binary mode. This keeps all the examples nice and simple and working consistently between Python 2 and Python 3. One of the big changes between Python 2 and Python 3 was that the default string type became unicode instead of binary, and a new type for bytes was introduced. This little snippet shows the difference by defining two constants for the British Pound symbol:

```
import sys
PY3 = sys.version_info[:2] >= (3, 0)

if PY3:
    some_bytes = '\xa3'.encode('utf-8')
    some_text = '\xa3'
else:
    some_bytes = '\xc2\xa3'
    some_text = '\xc2\xa3'.decode('utf-8')
```

Python 3 is much stricter than Python 2 about the byte versus string boundary and *TempDirectory* has been changed to help work with this by only reading and writing files in binary mode and providing parameters to control decoding and encoding when you want to read and write text.

For example, when writing, you can either write bytes directly, as we have been in the examples so far:

```
>>> path = tempdir.write('currencies.txt', some_bytes)
>>> with open(path, 'rb') as currencies:
...     currencies.read()
b'\xc2\xa3'
```

Or, you can write text, but must specify an encoding to use when writing the data to the file:

```
>>> path = tempdir.write('currencies.txt', some_text, 'utf-8')
>>> with open(path, 'rb') as currencies:
...     currencies.read()
b'\xc2\xa3'
```

The same is true when reading files. You can either read bytes:

```
>>> tempdir.read('currencies.txt') == some_bytes
True
```

Or, you can read text, but must specify an encoding that will be used to decode the data in the file:

```
>>> tempdir.read('currencies.txt', 'utf-8') == some_text
True
```

Working with an existing sandbox

Some testing infrastructure already provides a sandbox temporary directory, however that infrastructure might not provide the same level of functionality that *TempDirectory* provides.

For this reason, it is possible to wrap an existing directory such as the following with a *TempDirectory*:

```
>>> from tempfile import mkdtemp
>>> thedir = mkdtemp()
```

When working with the context manager, this is done as follows:

```
>>> with TempDirectory(path=thedir) as d:
...   d.write('file', b'data')
...   d.makedir('directory')
...   sorted(os.listdir(thedir))
'...'
'...'
['directory', 'file']
```

For the decorator, usage would be as follows:

```
from testfixtures import tempdir, compare

@tempdir(path=thedir)
def test_function(d):
    d.write('file', b'data')
    d.makedir('directory')
    assert sorted(os.listdir(thedir))==['directory', 'file']
```

It is important to note that if an existing directory is used, it will not be deleted by either the decorator or the context manager. You will need to make sure that the directory is cleaned up as required.

Using with Sybil

Sybil is a tool for testing the examples found in documentation. It works by applying a set of specialised parsers to the documentation and testing or otherwise using the examples returned by those parsers.

The key differences between testing with Sybil and traditional doctests are that it is possible to plug in different types of parser, not just the "python console session" one, and so it is possible to test different types of examples. TestFixtures provides one these parsers to aid working with *TempDirectory* objects. This parser makes use of `topic` directives with specific classes set to perform different actions.

The following sections describe how to use this parser to help with writing temporary files and checking their contents.

Setting up

To use the Sybil parser, you need to make sure a *TempDirectory* instance is available under a particular name in the sybil test namespace. This name is then passed to the parser's constructor and the parser is passed to the `Sybil` constructor.

The following example shows how to use Sybil's pytest integration to execute all of the examples below. These require not only the TestFixtures parser but also the Sybil parsers that give more traditional doctest behaviour, invisible code blocks that are useful for setting things up and checking examples without breaking up the flow of the documentation, and capturing of examples from the documentation to use for use in other forms of testing:

```
from sybil import Sybil
from sybil.parsers.doctest import  DocTestParser
from sybil.parsers.codeblock import CodeBlockParser
from sybil.parsers.capture import parse_captures

from testfixtures import TempDirectory
from testfixtures.sybil import FileParser


def sybil_setup(namespace):
    namespace['tempdir'] = TempDirectory()


def sybil_teardown(namespace):
    namespace['tempdir'].cleanup()


pytest_collect_file = Sybil(
    parsers=[
        DocTestParser(),
        CodeBlockParser(),
        parse_captures,
        FileParser('tempdir'),
    ],
    pattern='*.txt',
    setup=sybil_setup, teardown=sybil_teardown,
).pytest()
```

Writing files

To write a file, a `topic` with a class of `write-file` is included in the documentation. The following example is a complete reStructuredText file that shows how to write a file that is then used by a later example:

```
Here's an example configuration file:

.. topic:: example.cfg
 :class: write-file

```

```
 ::

   [A Section]
   dir=frob
   long: this value continues
     on the next line

.. invisible-code-block: python

  from testfixtures.compat import PY3
  # change to the temp directory
  import os
  original_dir = os.getcwd()
  os.chdir(tempdir.path)

To parse this file using the :mod:`ConfigParser` module, you would
do the following:

.. code-block:: python

   if PY3:
       from configparser import ConfigParser
   else:
       from ConfigParser import ConfigParser
   config = ConfigParser()
   config.read('example.cfg')

The items in the section are now available as follows:

>>> for name, value in sorted(config.items('A Section')):
...     print('{0!r}:{1!r}'.format(name, value))
'dir':'frob'
'long':'this value continues\non the next line'

.. invisible-code-block: python

  # change out again
  import os
  os.chdir(original_dir)
```

Checking the contents of files

To read a file, a `topic` with a class of `read-file` is included in the documentation. The following example is a complete reStructuredText file that shows how to check the values written by the code being documented while also using this check as part of the documentation:

```
.. invisible-code-block: python

  from testfixtures.compat import PY3
  # change to the temp directory
  import os
  original_dir = os.getcwd()
  os.chdir(tempdir.path)

To construct a configuration file using the :mod:`ConfigParser`
module, you would do the following:
```

```
.. code-block:: python

   if PY3:
       from configparser import ConfigParser
   else:
       from ConfigParser import ConfigParser
   config = ConfigParser()
   config.add_section('A Section')
   config.set('A Section', 'dir', 'frob')
   f = open('example.cfg','w')
   config.write(f)
   f.close()

The generated configuration file will be as follows:

.. topic:: example.cfg
 :class: read-file

 ::

   [A Section]
   dir = frob


.. config parser writes whitespace at the end, be careful when testing!

.. invisible-code-block: python

  # change out again
  import os
  os.chdir(original_dir)
```

Checking the contents of directories

While `FileParser` itself does not offer any facility for checking the contents of directories, Sybil's `parse_captures()` can be used in conjunction with the existing features of a *TempDirectory* to illustrate the contents expected in a directory seamlessly within the documentation.

Here's a complete reStructuredText document that illustrates this technique:

```
Here's an example piece of code that creates some files and
directories:

.. code-block:: python

  import os

  def spew(path):
    with open(os.path.join(path, 'root.txt'), 'wb') as f:
        f.write(b'root output')
    os.mkdir(os.path.join(path, 'subdir'))
    with open(os.path.join(path, 'subdir', 'file.txt'), 'wb') as f:
        f.write(b'subdir output')
    os.mkdir(os.path.join(path, 'subdir', 'logs'))
```

```
This function is used as follows:

>>> spew(tempdir.path)

This will create the following files and directories::

  root.txt
  subdir/
  subdir/file.txt
  subdir/logs/

.. -> expected_listing

.. invisible-code-block: python

  # check the listing was as expected
  tempdir.compare(expected_listing.strip().split('\n'))
```

A note on encoding and line endings

As currently implemented, the parser provided by TestFixtures only works with textual file content that can be encoded using the ASCII character set. This content will always be written with `'\n'` line separators and, when read, will always have its line endings normalised to `'\n'`. If you hit any limitations caused by this, please raise an issue in the tracker on GitHub.

CHAPTER 7

Testing exceptions

The `unittest` support for asserting that exceptions are raised when expected is fairly weak. Like many other Python testing libraries, TestFixtures has tools to help with this.

The `ShouldRaise` context manager

If you are using a version of Python where the `with` statement can be used, it's recommended that you use the *`ShouldRaise`* context manager.

Suppose we wanted to test the following function to make sure that the right exception was raised:

```
def the_thrower(throw=True):
    if throw:
        raise ValueError('Not good!')
```

The following example shows how to test that the correct exception is raised:

```
>>> from testfixtures import ShouldRaise
>>> with ShouldRaise(ValueError('Not good!')):
...     the_thrower()
```

If the exception raised doesn't match the one expected, *`ShouldRaise`* will raise an `AssertionError` causing the tests in which it occurs to fail:

```
>>> with ShouldRaise(ValueError('Is good!')):
...     the_thrower()
Traceback (most recent call last):
...
AssertionError: ValueError('Not good!',) raised, ValueError('Is good!',) expected
```

If you're not concerned about anything more than the type of the exception that's raised, you can check as follows:

```
>>> from testfixtures import ShouldRaise
>>> with ShouldRaise(ValueError):
...     the_thrower()
```

If you're feeling slack and just want to check that an exception is raised, but don't care about the type of that exception, the following will suffice:

```
>>> from testfixtures import ShouldRaise
>>> with ShouldRaise():
...     the_thrower()
```

If no exception is raised by the code under test, *ShouldRaise* will raise an AssertionError to indicate this:

```
>>> from testfixtures import ShouldRaise
>>> with ShouldRaise():
...     the_thrower(throw=False)
Traceback (most recent call last):
...
AssertionError: No exception raised!
```

ShouldRaise has been implemented such that it can be successfully used to test if code raises both SystemExit and KeyboardInterrupt exceptions.

To help with SystemExit and other exceptions that are tricky to construct yourself, *ShouldRaise* instances have a *raised* attribute. This will contain the actual exception raised and can be used to inspect parts of it:

```
>>> import sys
>>> from testfixtures import ShouldRaise
>>> with ShouldRaise() as s:
...     sys.exit(42)
>>> s.raised.code
42
```

The should_raise() decorator

If you are working in a traditional unittest environment and want to check that a particular test function raises an exception, you may find the decorator suits your needs better:

```
from testfixtures import should_raise

@should_raise(ValueError('Not good!'))
def test_function():
    the_thrower()
```

This decorator behaves exactly as the *ShouldRaise* context manager described in the documentation above.

Note: It is slightly recommended that you use the context manager rather than the decorator in most cases. With the decorator, all exceptions raised within the decorated function will be checked, which can hinder test development. With the context manager, you can make assertions about only the exact lines of code that you expect to raise the exception.

Exceptions that are conditionally raised

Some exceptions are only raised in certain versions of Python. For example, in Python 2, `bytes()` will turn both bytes and strings into bytes, while in Python 3, it will raise an exception when presented with a string. If you wish to make assertions that this behaviour is expected, you can use the `unless` option to *ShouldRaise* as follows:

```
import sys
from testfixtures import ShouldRaise

PY2 = sys.version_info[:2] <  (3, 0)

with ShouldRaise(TypeError, unless=PY2):
    bytes('something')
```

Note: Do **not** abuse this functionality to make sloppy assertions. It is always better have two different tests that cover a case when an exception should be raised and a case where an exception should not be raised rather than using it above functionality. It is *only* provided to help in cases where something in the environment that cannot be mocked out or controlled influences whether or not an exception is raised.

CHAPTER 8

Testing warnings

The `unittest` support for asserting that warnings are issued when expected is fairly convoluted, so TestFixtures has tools to help with this.

The `ShouldWarn` context manager

This context manager allows you to assert that particular warnings are recorded in a block of code, for example:

```
>>> from warnings import warn
>>> from testfixtures import ShouldWarn
>>> with ShouldWarn(UserWarning('you should fix that')):
...     warn('you should fix that')
```

If a warning issued doesn't match the one expected, *`ShouldWarn`* will raise an `AssertionError` causing the test in which it occurs to fail:

```
>>> from warnings import warn
>>> from testfixtures import ShouldWarn
>>> with ShouldWarn(UserWarning('you should fix that')):
...     warn("sorry dave, I can't let you do that")
Traceback (most recent call last):
...
AssertionError: sequence not as expected:

same:
[]

expected:
[
  <C(failed):....UserWarning>
  args:('you should fix that',) != ("sorry dave, I can't let you do that",)
  </C>]
```

```
actual:
[UserWarning("sorry dave, I can't let you do that",)]
```

You can check multiple warnings in a particular piece of code:

```
>>> from warnings import warn
>>> from testfixtures import ShouldWarn
>>> with ShouldWarn(UserWarning('you should fix that'),
...                 UserWarning('and that too')):
...     warn('you should fix that')
...     warn('and that too')
```

If you want to inspect more details of the warnings issued, you can capture them into a list as follows:

```
>>> from warnings import warn_explicit
>>> from testfixtures import ShouldWarn
>>> with ShouldWarn() as captured:
...     warn_explicit(message='foo', category=DeprecationWarning,
...                   filename='bar.py', lineno=42)
>>> len(captured)
1
>>> captured[0].message
DeprecationWarning('foo',)
>>> captured[0].lineno
42
```

The `ShouldNotWarn` context manager

If you do not expect any warnings to be logged in a piece of code, you can use the *`ShouldNotWarn`* context manager. If any warnings are issued in the context it manages, it will raise an `AssertionError` to indicate this:

```
>>> from warnings import warn
>>> from testfixtures import ShouldNotWarn
>>> with ShouldNotWarn():
...     warn("woah dude")
Traceback (most recent call last):
...
AssertionError: sequence not as expected:

same:
[]

expected:
[]

actual:
[UserWarning('woah dude',)]
```

CHAPTER 9

Testing use of the subprocess package

When using the `subprocess` package there are two approaches to testing:

- Have your tests exercise the real processes being instantiated and used.
- Mock out use of the `subprocess` package and provide expected output while recording interactions with the package to make sure they are as expected.

While the first of these should be preferred, it means that you need to have all the external software available everywhere you wish to run tests. Your tests will also need to make sure any dependencies of that software on an external environment are met. If that external software takes a long time to run, your tests will also take a long time to run.

These challenges can often make the second approach more practical and can be the more pragmatic approach when coupled with a mock that accurately simulates the behaviour of a subprocess. *`MockPopen`* is an attempt to provide just such a mock.

Note: To use *`MockPopen`*, you must have the `mock` package installed.

Example usage

As an example, suppose you have code such as the following that you need to test:

```
from subprocess import Popen, PIPE


def my_func():
    process = Popen('svn ls -R foo', stdout=PIPE, stderr=PIPE, shell=True)
    out, err = process.communicate()
    if process.returncode:
        raise RuntimeError('something bad happened')
    return out
```

Tests that exercises this code using *`MockPopen`* could be written as follows:

```
from unittest import TestCase

from mock import call
from testfixtures import Replacer, ShouldRaise, compare
from testfixtures.popen import MockPopen


class TestMyFunc(TestCase):

    def setUp(self):
        self.Popen = MockPopen()
        self.r = Replacer()
        self.r.replace(dotted_path, self.Popen)
        self.addCleanup(self.r.restore)

    def test_example(self):
        # set up
        self.Popen.set_command('svn ls -R foo', stdout=b'o', stderr=b'e')

        # testing of results
        compare(my_func(), b'o')

        # testing calls were in the right order and with the correct parameters:
        compare([
            call.Popen('svn ls -R foo',
                       shell=True, stderr=PIPE, stdout=PIPE),
            call.Popen_instance.communicate()
            ], Popen.mock.method_calls)

    def test_example_bad_returncode(self):
        # set up
        Popen.set_command('svn ls -R foo', stdout=b'o', stderr=b'e',
                          returncode=1)

        # testing of error
        with ShouldRaise(RuntimeError('something bad happened')):
            my_func()
```

Passing input to processes

If your testing requires passing input to the subprocess, you can do so by checking for the input passed to `communicate()` method when you check the calls on the mock as shown in this example:

```
def test_communicate_with_input(self):
    # setup
    Popen = MockPopen()
    Popen.set_command('a command')
    # usage
    process = Popen('a command', stdout=PIPE, stderr=PIPE, shell=True)
    out, err = process.communicate('foo')
    # test call list
    compare([
            call.Popen('a command', shell=True, stderr=-1, stdout=-1),
            call.Popen_instance.communicate('foo'),
            ], Popen.mock.method_calls)
```

Note: Accessing `.stdin` isn't current supported by this mock.

Reading from `stdout` and `stderr`

The `.stdout` and `.stderr` attributes of the mock returned by *MockPopen* will be file-like objects as with the real `Popen` and can be read as shown in this example:

```
def test_read_from_stdout_and_stderr(self):
    # setup
    Popen = MockPopen()
    Popen.set_command('a command', stdout=b'foo', stderr=b'bar')
    # usage
    process = Popen('a command', stdout=PIPE, stderr=PIPE, shell=True)
    compare(process.stdout.read(), b'foo')
    compare(process.stderr.read(), b'bar')
    # test call list
    compare([
            call.Popen('a command', shell=True, stderr=PIPE, stdout=PIPE),
            ], Popen.mock.method_calls)
```

Warning: While these streams behave a lot like the streams of a real `Popen` object, they do not exhibit the deadlocking behaviour that can occur when the two streams are read as in the example above. Be very careful when reading `.stdout` and `.stderr` and consider using `communicate` instead.

Specifying the return code

Often code will need to behave differently depending on the return code of the launched process. Specifying a simulated response code, along with testing for the correct usage of `wait()`, can be seen in the following example:

```
def test_wait_and_return_code(self):
    # setup
    Popen = MockPopen()
    Popen.set_command('a command', returncode=3)
    # usage
    process = Popen('a command')
    compare(process.returncode, None)
    # result checking
    compare(process.wait(), 3)
    compare(process.returncode, 3)
    # test call list
    compare([
            call.Popen('a command'),
            call.Popen_instance.wait(),
            ], Popen.mock.method_calls)
```

Checking for signal sending

Calls to `.send_signal()`, `.terminate()` and `.kill()` are all recorded by the mock returned by *MockPopen* but otherwise do nothing as shown in the following example, which doesn't make sense for a real test of sub-process usage but does show how the mock behaves:

```
def test_send_signal(self):
    # setup
    Popen = MockPopen()
    Popen.set_command('a command')
    # usage
    process = Popen('a command', stdout=PIPE, stderr=PIPE, shell=True)
    process.send_signal(0)
    # result checking
    compare([
            call.Popen('a command', shell=True, stderr=-1, stdout=-1),
            call.Popen_instance.send_signal(0),
            ], Popen.mock.method_calls)
```

Polling a process

The `poll()` method is often used as part of a loop in order to do other work while waiting for a sub-process to complete. The mock returned by *MockPopen* supports this by allowing the `.poll()` method to be called a number of times before the `returncode` is set using the `poll_count` parameter as shown in the following example:

```
def test_poll_until_result(self):
    # setup
    Popen = MockPopen()
    Popen.set_command('a command', returncode=3, poll_count=2)
    # example usage
    process = Popen('a command')
    while process.poll() is None:
        # you'd probably have a sleep here, or go off and
        # do some other work.
        pass
    # result checking
    compare(process.returncode, 3)
    compare([
            call.Popen('a command'),
            call.Popen_instance.poll(),
            call.Popen_instance.poll(),
            call.Popen_instance.poll(),
            ], Popen.mock.method_calls)
```

Using default behaviour

If you're testing something that needs to make many calls to many different commands that all behave the same, it can be tedious to specify the behaviour of each with *set_command*. For this case, *MockPopen* has the *set_default* method which can be used to set the behaviour of any command that has not been specified with *set_command* as shown in the following example:

```
def test_default_behaviour(self):
    # set up
    self.Popen.set_default(stdout=b'o', stderr=b'e')

    # testing of results
    compare(my_func(), b'o')

    # testing calls were in the right order and with the correct parameters:
    compare([
        call.Popen('svn ls -R foo',
                   shell=True, stderr=PIPE, stdout=PIPE),
        call.Popen_instance.communicate()
        ], Popen.mock.method_calls)
```

CHAPTER 10

Testing when using django

Django's ORM has an unfortunate implementation choice to consider `Model` instances to be identical as long as their primary keys are the same:

```
>>> from testfixtures.tests.test_django.models import SampleModel
>>> SampleModel(id=1, value=1) == SampleModel(id=1, value=2)
True
```

To work around this, `testfixtures.django` *registers* a `comparer` for the django `Model` class. However, for this to work, `ignore_eq=True` must be passed:

```
>>> from testfixtures import compare
>>> import testfixtures.django # to register the comparer...
>>> compare(SampleModel(id=1, value=1), SampleModel(id=1, value=2),
...         ignore_eq=True)
Traceback (most recent call last):
 ...
AssertionError: SampleModel not as expected:

same:
[u'id']

values differ:
'value': 1 != 2
```

Since the above can quickly become cumbersome, a django-specific version of *`compare()`*, with ignoring `__eq__` built in, is provided:

```
>>> from testfixtures.django import compare as django_compare
>>> django_compare(SampleModel(id=1, value=1), SampleModel(id=1, value=2))
Traceback (most recent call last):
 ...
AssertionError: SampleModel not as expected:

same:
[u'id']
```

```
values differ:
'value': 1 != 2
```

It may also be that you want to ignore fields over which you have no control and cannot easily mock, such as created or modified times. For this, you can use the *ignore_fields* option:

```
>>> compare(SampleModel(id=1, value=1), SampleModel(id=1, value=2),
...         ignore_eq=True, ignore_fields=['value'])
```

Note: The implementation of the comparer for `Model` instances ignores fields that have `editable` set to `False`.

By default, non-editable fields are ignored:

```
>>> django_compare(SampleModel(not_editable=1), SampleModel(not_editable=2))
```

If you wish to include these fields in the comparison, pass the `non_editable_fields` option:

```
>>> django_compare(SampleModel(not_editable=1), SampleModel(not_editable=2),
...                non_editable_fields=True)
Traceback (most recent call last):
  ...
AssertionError: SampleModel not as expected:

same:
['created', u'id', 'value']

values differ:
'not_editable': 1 != 2
```

CHAPTER 11

Testing with zope.component

zope.component is a fantastic aspect-oriented library for Python, however its unit testing support is somewhat convoluted. If you need to test code that registers adapters, utilities and the like then you may need to provide a sterile component registry. For historical reasons, component registries are known as *Site Managers* in `zope.component`.

TestFixtures provides the a `TestComponents` helper which provides just such a sterile registry. It should be instantiated in your `TestCase`'s `setUp()` method. It's `uninstall()` method should be called in the test's `tearDown()` method.

Normally, `zope.component.getSiteManager()` returns whatever the current registry is. This may be influenced by frameworks that use `zope.component` which can means that unit tests have no baseline to start with:

```
>>> from zope.component import getSiteManager
>>> original = getSiteManager()
>>> print(original)
<BaseGlobalComponents base>
```

Once we've got a `TestComponents` in place, we know what we're getting:

```
>>> from testfixtures.components import TestComponents
>>> components = TestComponents()
>>> getSiteManager()
<Components Testing>
```

The registry that `getSiteManager()` returns is now also available as an attribute of the `TestComponents` instance:

```
>>> getSiteManager() is components.registry
True
```

It's also empty:

```
>>> tuple(components.registry.registeredUtilities())
()
>>> tuple(components.registry.registeredAdapters())
()
```

```
>>> tuple(components.registry.registeredHandlers())
()
```

You can do whatever you like with this registry. When you're done, just call the `uninstall()` method:

```
>>> components.uninstall()
```

Now you'll have the original registy back in place:

```
>>> getSiteManager() is original
True
```

CHAPTER 12

Utilities

This section describes a few handy functions that didn't fit nicely in any other section.

The `generator` helper

It can be handy when testing to be able to turn a simple sequence into a generator. This can be necessary when you want to check that your code will behave correctly when processing a generator instead of a simple sequence, or when you're looking to make assertions about the expected return value of a callable that returns a generator.

If you need to turn a simple sequence into a generator, the *`generator()`* function is the way to do it:

```
>>> from testfixtures import generator
>>> generator(1,2,3)
<generator object ...>
```

Iterating over this generator will return the arguments passed to the *`generator()`* function:

```
>>> for i in _:
...     print(i, end=' ')
1 2 3
```

The `wrap` helper

The *`wrap()`* helper is a decorator function that allows you to wrap the call to the decorated callable with calls to other callables. This can be useful when you want to perform setup and teardown actions either side of a test function.

For example, take the following functions:

```
def before():
    print("before")
```

```
def after():
    print("after")
```

The *wrap()* helper can be used to wrap a function with these:

```
from testfixtures import wrap

@wrap(before,after)
def a_function():
    print("a_function")
```

When the wrapped function is executed, the output is as follows:

```
>>> a_function()
before
a_function
after
```

The section argument to *wrap()* is optional:

```
from testfixtures import wrap

@wrap(before)
def a_function():
    print("a_function")
```

Now, the wrapped function gives the following output when executed:

```
>>> a_function()
before
a_function
```

Multiple wrapping functions can be provided by stacking *wrap()* decorations:

```
def before1():
    print("before 1")

def after1():
    print("after 1")

def before2():
    print("before 2")

def after2():
    print("after 2")

@wrap(before2,after2)
@wrap(before1,after1)
def a_function():
    print("a_function")
```

The order of execution is illustrated below:

```
>>> a_function()
before 1
before 2
a_function
```

```
after 2
after 1
```

The results of calling the wrapping functions executed before the wrapped function can be made available to the wrapped function provided it accepts positional arguments for these results:

```
def before1():
    return "return 1"

def before2():
    return "return 2"

@wrap(before2)
@wrap(before1)
def a_function(r1,r2):
    print(r1)
    print(r2)
```

Calling the wrapped function illustrates the behaviour:

```
>>> a_function()
return 1
return 2
```

Finally, the return value of the wrapped function will always be that of the original function:

```
def before1():
    return 1

def after1():
    return 2

def before2():
    return 3

def after2():
    return 4

@wrap(before2,after2)
@wrap(before1,after2)
def a_function():
    return 'original'
```

When the above wrapped function is executed, the original return value is still returned:

```
>>> a_function()
'original'
```

If you're looking for a description of a particular tool, please see the API reference:

CHAPTER 13

API Reference

class testfixtures.**Comparison**(*object_or_type*, *attribute_dict=None*, *strict=True*, ***attributes*)

These are used when you need to compare objects that do not natively support comparison.

Parameters

- **object_or_type** – The object or class from which to create the *Comparison*.
- **attribute_dict** – An optional dictionary containing attibutes to place on the *Comparison*.
- **strict** – If true, any expected attributes not present or extra attributes not expected on the object involved in the comparison will cause the comparison to fail.
- **attributes** – Any other keyword parameters passed will placed as attributes on the *Comparison*.

class testfixtures.**LogCapture**(*names=None*, *install=True*, *level=1*, *propagate=None*, *attributes=('name'*, *'levelname'*, *'getMessage')*, *recursive_check=False*)

These are used to capture entries logged to the Python logging framework and make assertions about what was logged.

Parameters

- **names** – A string (or tuple of strings) containing the dotted name(s) of loggers to capture. By default, the root logger is captured.
- **install** – If *True*, the *LogCapture* will be installed as part of its instantiation.
- **propagate** – If specified, any captured loggers will have their *propagate* attribute set to the supplied value. This can be used to prevent propagation from a child logger to a parent logger that has configured handlers.
- **attributes** – The sequence of attribute names to return for each record or a callable that extracts a row from a record..

  If a sequence of attribute names, those attributes will be taken from the LogRecord. If an attribute is callable, the value used will be the result of calling it. If an attribute is missing,

None will be used in its place.

If a callable, it will be called with the LogRecord and the value returned will be used as the row..

- **recursive_check** – If True, log messages will be compared recursively by *LogCapture.check()*.

check(**expected*)
This will compare the captured entries with the expected entries provided and raise an AssertionError if they do not match.

Parameters **expected** – A sequence of 3-tuples containing the expected log entries. Each tuple should be of the form (logger_name, string_level, message)

clear()
Clear any entries that have been captured.

install()
Install this LogHandler into the Python logging framework for the named loggers.

This will remove any existing handlers for those loggers and drop their level to that specified on this *LogCapture* in order to capture all logging.

uninstall()
Un-install this LogHandler from the Python logging framework for the named loggers.

This will re-instate any existing handlers for those loggers that were removed during installation and retore their level that prior to installation.

classmethod **uninstall_all**()
This will uninstall all existing LogHandler objects.

class testfixtures.**OutputCapture**(*separate=False*)
A context manager for capturing output to the sys.stdout and sys.stderr streams.

Parameters **separate** – If True, stdout and stderr will be captured separately and their expected values must be passed to *compare()*.

Note: If separate is passed as True, *OutputCapture.captured* will be an empty string.

---

captured
A property containing any output that has been captured so far.

compare(*expected=''*, *stdout=''*, *stderr=''*)
Compare the captured output to that expected. If the output is not the same, an AssertionError will be raised.

Parameters

- **expected** – A string containing the expected combined output of stdout and stderr.
- **stdout** – A string containing the expected output to stdout.
- **stderr** – A string containing the expected output to stderr.

disable()
Disable the output capture if it is enabled.

enable()
Enable the output capture if it is disabled.

class testfixtures.**Replace**(*target*, *replacement*, *strict=True*)
A context manager that uses a *Replacer* to replace a single target.

Parameters

- **target** – A string containing the dotted-path to the object to be replaced. This path may specify a module in a package, an attribute of a module, or any attribute of something contained within a module.
- **replacement** – The object to use as a replacement.
- **strict** – When *True*, an exception will be raised if an attempt is made to replace an object that does not exist.

class testfixtures.**Replacer**
These are used to manage the mocking out of objects so that units of code can be tested without having to rely on their normal dependencies.

replace(*target*, *replacement*, *strict=True*)
Replace the specified target with the supplied replacement.

Parameters

- **target** – A string containing the dotted-path to the object to be replaced. This path may specify a module in a package, an attribute of a module, or any attribute of something contained within a module.
- **replacement** – The object to use as a replacement.
- **strict** – When *True*, an exception will be raised if an attempt is made to replace an object that does not exist.

restore()
Restore all the original objects that have been replaced by calls to the *replace()* method of this *Replacer*.

testfixtures.**replace**(*target*, *replacement*, *strict=True*)
A decorator to replace a target object for the duration of a test function.

Parameters

- **target** – A string containing the dotted-path to the object to be replaced. This path may specify a module in a package, an attribute of a module, or any attribute of something contained within a module.
- **replacement** – The object to use as a replacement.
- **strict** – When *True*, an exception will be raised if an attempt is made to replace an object that does not exist.

class testfixtures.**RoundComparison**(*value*, *precision*)
An object that can be used in comparisons of expected and actual numerics to a specified precision.

Parameters

- **value** – numeric to be compared.
- **precision** – Number of decimal places to round to in order to perform the comparison.

class testfixtures.**RangeComparison**(*lower_bound*, *upper_bound*)
An object that can be used in comparisons of orderable types to check that a value specified within the given range.

Parameters

- **lower_bound** – the inclusive lower bound for the acceptable range.
- **upper_bound** – the inclusive upper bound for the acceptable range.

class testfixtures.**ShouldRaise**(*exception=None*, *unless=False*)
This context manager is used to assert that an exception is raised within the context it is managing.

Parameters

- **exception** – This can be one of the following:
  - *None*, indicating that an exception must be raised, but the type is unimportant.
  - An exception class, indicating that the type of the exception is important but not the parameters it is created with.
  - An exception instance, indicating that an exception exactly matching the one supplied should be raised.
- **unless** – Can be passed a boolean that, when `True` indicates that no exception is expected. This is useful when checking that exceptions are only raised on certain versions of Python.

raised = None
The exception captured by the context manager. Can be used to inspect specific attributes of the exception.

class testfixtures.**ShouldWarn**(**expected*)
This context manager is used to assert that warnings are issued within the context it is managing.

Parameters **expected** –

This should be a sequence made up of one or more elements, each of one of the following types:

- A warning class, indicating that the type of the warnings is important but not the parameters it is created with.
- A warning instance, indicating that a warning exactly matching the one supplied should have been issued.

If no expected warnings are passed, you will need to inspect the contents of the list returned by the context manager.

class testfixtures.**ShouldNotWarn**
This context manager is used to assert that no warnings are issued within the context it is managing.

class testfixtures.**StringComparison**(*regex_source*)
An object that can be used in comparisons of expected and actual strings where the string expected matches a pattern rather than a specific concrete string.

Parameters **regex_source** – A string containing the source for a regular expression that will be used whenever this *StringComparison* is compared with any `basestring` instance.

class testfixtures.**TempDirectory**(*ignore=()*, *create=True*, *path=None*, *encoding=None*)
A class representing a temporary directory on disk.

Parameters

- **ignore** – A sequence of strings containing regular expression patterns that match filenames that should be ignored by the *TempDirectory* listing and checking methods.
- **create** – If *True*, the temporary directory will be created as part of class instantiation.
- **path** – If passed, this should be a string containing a physical path to use as the temporary directory. When passed, *TempDirectory* will not create a new directory to use.

- **encoding** – A default encoding to use for *read()* and *write()* operations when the `encoding` parameter is not passed to those methods.

check(**expected*)
Deprecated since version 4.3.0.

Compare the contents of the temporary directory with the expected contents supplied.

This method only checks the root of the temporary directory.

Parameters **expected** – A sequence of strings containing the names expected in the directory.

check_all(*dir*, **expected*)
Deprecated since version 4.3.0.

Recursively compare the contents of the specified directory with the expected contents supplied.

Parameters

- **dir** – The directory to check, which can be:
  - A tuple of strings, indicating that the elements of the tuple should be used as directory names to traverse from the root of the temporary directory to find the directory to be checked.
  - A forward-slash separated string, indicating the directory or subdirectory that should be traversed to from the temporary directory and checked.
  - An empty string, indicating that the whole temporary directory should be checked.
- **expected** – A sequence of strings containing the paths expected in the directory. These paths should be forward-slash separated and relative to the root of the temporary directory.

check_dir(*dir*, **expected*)
Deprecated since version 4.3.0.

Compare the contents of the specified subdirectory of the temporary directory with the expected contents supplied.

This method will only check the contents of the subdirectory specified and will not recursively check subdirectories.

Parameters

- **dir** – The subdirectory to check, which can be:
  - A tuple of strings, indicating that the elements of the tuple should be used as directory names to traverse from the root of the temporary directory to find the directory to be checked.
  - A forward-slash separated string, indicating the directory or subdirectory that should be traversed to from the temporary directory and checked.
- **expected** – A sequence of strings containing the names expected in the directory.

cleanup()
Delete the temporary directory and anything in it. This *TempDirectory* cannot be used again unless *create()* is called.

classmethod **cleanup_all**()
Delete all temporary directories associated with all *TempDirectory* objects.

compare(*expected*, *path=None*, *files_only=False*, *recursive=True*, *followlinks=False*)
Compare the expected contents with the actual contents of the temporary directory. An `AssertionError` will be raised if they are not the same.

Parameters

- **expected** – A sequence of strings containing the paths expected in the directory. These paths should be forward-slash separated and relative to the root of the temporary directory.
- **path** – The path to use as the root for the comparison, relative to the root of the temporary directory. This can either be:
  - A tuple of strings, making up the relative path.
  - A forward-slash separated string.

  If it is not provided, the root of the temporary directory will be used.
- **files_only** – If specified, directories will be excluded from the list of actual paths used in the comparison.
- **recursive** – If passed as `False`, only the direct contents of the directory specified by `path` will be included in the actual contents used for comparison.
- **followlinks** – If passed as `True`, symlinks and hard links will be followed when recursively building up the actual list of directory contents.

create()
Create a temporary directory for this instance to use if one has not already been created.

getpath(*path*)
Return the full path on disk that corresponds to the path relative to the temporary directory that is passed in.

Parameters **path** – The path to the file to create, which can be:

- A tuple of strings.
- A forward-slash separated string.

Returns A string containing the full path.

listdir(*path=None*, *recursive=False*)
Print the contents of the specified directory.

Parameters

- **path** – The path to list, which can be:
  - *None*, indicating the root of the temporary directory should be listed.
  - A tuple of strings, indicating that the elements of the tuple should be used as directory names to traverse from the root of the temporary directory to find the directory to be listed.
  - A forward-slash separated string, indicating the directory or subdirectory that should be traversed to from the temporary directory and listed.
- **recursive** – If *True*, the directory specified will have its subdirectories recursively listed too.

makedir(*dirpath*)
Make an empty directory at the specified path within the temporary directory. Any intermediate subdirectories that do not exist will also be created.

Parameters **dirpath** – The directory to create, which can be:

- A tuple of strings.
- A forward-slash separated string.

Returns The full path of the created directory.

path = None
The physical path of the *TempDirectory* on disk

read(*filepath*, *encoding=None*)
Reads the file at the specified path within the temporary directory.

The file is always read in binary mode. Bytes will be returned unless an encoding is supplied, in which case a unicode string of the decoded data will be returned.

Parameters

- **filepath** – The path to the file to read, which can be:
  - A tuple of strings.
  - A forward-slash separated string.
- **encoding** – The encoding used to decode the data in the file.

Returns A string containing the data read.

write(*filepath*, *data*, *encoding=None*)
Write the supplied data to a file at the specified path within the temporary directory. Any subdirectories specified that do not exist will also be created.

The file will always be written in binary mode. The data supplied must either be bytes or an encoding must be supplied to convert the string into bytes.

Parameters

- **filepath** – The path to the file to create, which can be:
  - A tuple of strings.
  - A forward-slash separated string.
- **data** – A string containing the data to be written.
- **encoding** – The encoding to be used if data is not bytes. Should not be passed if data is already bytes.

Returns The full path of the file written.

testfixtures.**compare**(*x*, *y*, *prefix=None*, *suffix=None*, *raises=True*, *recursive=True*, *strict=False*, *comparers=None*, ***kw*)
Compare the two arguments passed either positionally or using explicit `expected` and `actual` keyword paramaters. An `AssertionError` will be raised if they are not the same. The `AssertionError` raised will attempt to provide descriptions of the differences found.

Any other keyword parameters supplied will be passed to the functions that end up doing the comparison. See the API documentation below for details of these.

Parameters

- **prefix** – If provided, in the event of an `AssertionError` being raised, the prefix supplied will be prepended to the message in the `AssertionError`.
- **suffix** – If provided, in the event of an `AssertionError` being raised, the suffix supplied will be appended to the message in the `AssertionError`.
- **raises** – If `False`, the message that would be raised in the `AssertionError` will be returned instead of the exception being raised.
- **recursive** – If `True`, when a difference is found in a nested data structure, attempt to highlight the location of the difference.

- **strict** – If `True`, objects will only compare equal if they are of the same type as well as being equal.
- **ignore_eq** – If `True`, object equality, which relies on `__eq__` being correctly implemented, will not be used. Instead, comparers will be looked up and used and, if no suitable comparer is found, objects will be considered equal if their hash is equal.
- **comparers** – If supplied, should be a dictionary mapping types to comparer functions for those types. These will be added to the global comparer registry for the duration of this call.

`testfixtures.comparison.`**register**(*type*, *comparer*)
Register the supplied comparer for the specified type. This registration is global and will be in effect from the point this function is called until the end of the current process.

`testfixtures.comparison.`**compare_simple**(*x*, *y*, *context*)
Returns a very simple textual difference between the two supplied objects.

`testfixtures.comparison.`**compare_with_type**(*x*, *y*, *context*)
Return a textual description of the difference between two objects including information about their types.

`testfixtures.comparison.`**compare_sequence**(*x*, *y*, *context*)
Returns a textual description of the differences between the two supplied sequences.

`testfixtures.comparison.`**compare_generator**(*x*, *y*, *context*)
Returns a textual description of the differences between the two supplied generators.

This is done by first unwinding each of the generators supplied into tuples and then passing those tuples to `compare_sequence()`.

`testfixtures.comparison.`**compare_tuple**(*x*, *y*, *context*)
Returns a textual difference between two tuples or `collections.namedtuple()` instances.

The presence of a `_fields` attribute on a tuple is used to decide whether or not it is a `namedtuple()`.

`testfixtures.comparison.`**compare_dict**(*x*, *y*, *context*)
Returns a textual description of the differences between the two supplied dictionaries.

`testfixtures.comparison.`**compare_set**(*x*, *y*, *context*)
Returns a textual description of the differences between the two supplied sets.

`testfixtures.comparison.`**compare_text**(*x*, *y*, *context*)
Returns an informative string describing the differences between the two supplied strings. The way in which this comparison is performed can be controlled using the following parameters:

Parameters

- **blanklines** – If *False*, then when comparing multi-line strings, any blank lines in either argument will be ignored.
- **trailing_whitespace** – If *False*, then when comparing multi-line strings, trailing whilespace on lines will be ignored.
- **show_whitespace** – If *True*, then whitespace characters in multi-line strings will be replaced with their representations.

`testfixtures.`**diff**(*x*, *y*, *x_label=''*, *y_label=''*)
A shorthand function that uses `difflib` to return a string representing the differences between the two string arguments.

Most useful when comparing multi-line strings.

`testfixtures.`**generator**(**args*)
A utility function for creating a generator that will yield the supplied arguments.

testfixtures.**log_capture**(**names*, ***kw*)

A decorator for making a *LogCapture* installed an available for the duration of a test function.

Parameters **names** – An optional sequence of names specifying the loggers to be captured. If not specified, the root logger will be captured.

Keyword parameters other than `install` may also be supplied and will be passed on to the *LogCapture* constructor.

class testfixtures.**should_raise**(*exception=None*, *unless=None*)

A decorator to assert that the decorated function will raised an exception. An exception class or exception instance may be passed to check more specifically exactly what exception will be raised.

Parameters

- **exception** – This can be one of the following:
  - *None*, indicating that an exception must be raised, but the type is unimportant.
  - An exception class, indicating that the type of the exception is important but not the parameters it is created with.
  - An exception instance, indicating that an exception exactly matching the one supplied should be raised.
- **unless** – Can be passed a boolean that, when `True` indicates that no exception is expected. This is useful when checking that exceptions are only raised on certain versions of Python.

testfixtures.**tempdir**(**args*, ***kw*)

A decorator for making a *TempDirectory* available for the duration of a test function.

All arguments and parameters are passed through to the *TempDirectory* constructor.

testfixtures.**test_date**(*year=2001*, *month=1*, *day=1*, *delta=None*, *delta_type='days'*, *strict=False*)

A function that returns a mock object that can be used in place of the `datetime.date` class but where the return value of `today()` can be controlled.

If a single positional argument of `None` is passed, then the queue of dates to be returned will be empty and you will need to call *set()* or *add()* before calling *today()*.

Parameters

- **year** – An optional year used to create the first date returned by `today()`.
- **month** – An optional month used to create the first date returned by `today()`.
- **day** – An optional day used to create the first date returned by `today()`.
- **delta** – The size of the delta to use between values returned from `today()`. If not specified, it will increase by 1 with each call to `today()`.
- **delta_type** – The type of the delta to use between values returned from `today()`. This can be any keyword parameter accepted by the `timedelta` constructor.
- **strict** – If `True`, calling the mock class and any of its methods will result in an instance of the mock being returned. If `False`, the default, an instance of `date` will be returned instead.

The mock returned will behave exactly as the `datetime.date` class with the exception of the following members:

tdate.**add**(**args*, ***kw*)

This will add the `datetime.date` created from the supplied parameters to the queue of dates to be returned by *today()*. An instance of `date` may also be passed as a single positional argument.

tdate.**set**(**args*, ***kw*)
This will set the datetime.date created from the supplied parameters as the next date to be returned by *today()*, regardless of any dates in the queue. An instance of date may also be passed as a single positional argument.

tdate.**tick**(**args*, ***kw*)
This method should be called either with a timedelta as a positional argument, or with keyword parameters that will be used to construct a timedelta.

The timedelta will be used to advance the next date to be returned by *today()*.

classmethod tdate.**today**()
This will return the next supplied or calculated date from the internal queue, rather than the actual current date.

testfixtures.**test_datetime**(*year=2001*, *month=1*, *day=1*, *hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, *delta=None*, *delta_type='seconds'*, *date_type=datetime.date*, *strict=False*)
A function that returns a mock object that can be used in place of the datetime.datetime class but where the return value of *now()* can be controlled.

If a single positional argument of None is passed, then the queue of datetimes to be returned will be empty and you will need to call *set()* or *add()* before calling *now()* or *utcnow()*.

Parameters

- **year** – An optional year used to create the first datetime returned by *now()*.
- **month** – An optional month used to create the first datetime returned by *now()*.
- **day** – An optional day used to create the first datetime returned by *now()*.
- **hour** – An optional hour used to create the first datetime returned by *now()*.
- **minute** – An optional minute used to create the first datetime returned by *now()*.
- **second** – An optional second used to create the first datetime returned by *now()*.
- **microsecond** – An optional microsecond used to create the first datetime returned by *now()*.
- **tzinfo** – An optional tzinfo that will be used to indicate the timezone intended for the values returned by returned by *now()*. It will be used to correctly calculate return values when *tz* is passed to *now()* and when *utcnow()* is called.
- **delta** – The size of the delta to use between values returned from *now()*. If not specified, it will increase by 1 with each call to *now()*.
- **delta_type** – The type of the delta to use between values returned from *now()*. This can be any keyword parameter accepted by the timedelta constructor.
- **date_type** – The type to use for the return value of the date() method. This can help with gotchas that occur when type checking if performed on values returned by the mock's date() method.
- **strict** – If True, calling the mock class and any of its methods will result in an instance of the mock being returned. If False, the default, an instance of datetime will be returned instead.

The mock returned will behave exactly as the datetime.datetime class with the exception of the following members:

tdatetime.**add**(**args*, ***kw*)
This will add the datetime.datetime created from the supplied parameters to the queue of datetimes

to be returned by *now()* or *utcnow()*. An instance of datetime may also be passed as a single positional argument.

tdatetime.**set**(**args*, **kw*)
This will set the datetime.datetime created from the supplied parameters as the next datetime to be returned by *now()* or *utcnow()*, clearing out any datetimes in the queue. An instance of datetime may also be passed as a single positional argument.

tdatetime.**tick**(**args*, ***kw*)
This method should be called either with a timedelta as a positional argument, or with keyword parameters that will be used to construct a timedelta.

The timedelta will be used to advance the next datetime to be returned by *now()* or *utcnow()*.

classmethod tdatetime.**now**([*tz*])

Parameters **tz** – An optional timezone to apply to the returned time. If supplied, it must be an instance of a tzinfo subclass.

This will return the next supplied or calculated datetime from the internal queue, rather than the actual current datetime.

If *tz* is supplied, it will be applied to the datetime that would have have been returned from the internal queue, treating that datetime as if it were in the UTC timezone.

classmethod tdatetime.**utcnow**()
This will return the next supplied or calculated datetime from the internal queue, rather than the actual current UTC datetime.

No timezone will be applied, even that supplied to the constructor.

classmethod tdatetime.**date**()
This will return the date component of the current mock instance, but using the date type supplied when the mock class was created.

testfixtures.**test_time**(*year=2001*, *month=1*, *day=1*, *hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, *delta=None*, *delta_type='seconds'*)
A function that returns a mock object that can be used in place of the time.time function but where the return value can be controlled.

If a single positional argument of None is passed, then the queue of times to be returned will be empty and you will need to call *set()* or *add()* before calling the mock.

Parameters

- **year** – An optional year used to create the first time returned.
- **month** – An optional month used to create the first time.
- **day** – An optional day used to create the first time.
- **hour** – An optional hour used to create the first time.
- **minute** – An optional minute used to create the first time.
- **second** – An optional second used to create the first time.
- **microsecond** – An optional microsecond used to create the first time.
- **delta** – The size of the delta to use between values returned. If not specified, it will increase by 1 with each call to the mock.
- **delta_type** – The type of the delta to use between values returned. This can be any keyword parameter accepted by the timedelta constructor.

The mock additionally has the following methods available on it:

ttime.**add**(**args*, ***kw*)
This will add the time specified by the supplied parameters to the queue of times to be returned by calls to the mock. The parameters are the same as the `datetime.datetime` constructor. An instance of `datetime` may also be passed as a single positional argument.

ttime.**set**(**args*, ***kw*)
This will set the time specified by the supplied parameters as the next time to be returned by a call to the mock, regardless of any times in the queue. The parameters are the same as the `datetime.datetime` constructor. An instance of `datetime` may also be passed as a single positional argument.

ttime.**tick**(**args*, ***kw*)
This method should be called either with a `timedelta` as a positional argument, or with keyword parameters that will be used to construct a `timedelta`.

The `timedelta` will be used to advance the next time to be returned by a call to the mock.

testfixtures.**wrap**(*before*, *after=None*)
A decorator that causes the supplied callables to be called before or after the wrapped callable, as appropriate.

testfixtures.**not_there**
A singleton used to represent the absence of a particular attribute.

class testfixtures.popen.**MockPopen**
A specialised mock for testing use of `subprocess.Popen`. An instance of this class can be used in place of the `subprocess.Popen` and is often inserted where it's needed using `mock.patch()` or a `Replacer`.

communicate(*input=None*)
Simulate calls to `subprocess.Popen.communicate()`

kill()
Simulate calls to `subprocess.Popen.kill()`

poll()
Simulate calls to `subprocess.Popen.poll()`

send_signal(*signal*)
Simulate calls to `subprocess.Popen.send_signal()`

set_command(*command*, *stdout=b''*, *stderr=b''*, *returncode=0*, *pid=1234*, *poll_count=3*)
Set the behaviour of this mock when it is used to simulate the specified command.

Parameters

- **command** – A string representing the command to be simulated.
- **stdout** – A string representing the simulated content written by the process to the stdout pipe.
- **stderr** – A string representing the simulated content written by the process to the stderr pipe.
- **returncode** – An integer representing the return code of the simulated process.
- **pid** – An integer representing the process identifier of the simulated process. This is useful if you have code the prints out the pids of running processes.
- **poll_count** – Specifies the number of times *`MockPopen.poll()`* can be called before `MockPopen.returncode` is set and returned by *`MockPopen.poll()`*.

set_default(*stdout=b''*, *stderr=b''*, *returncode=0*, *pid=1234*, *poll_count=3*)
Set the behaviour of this mock when it is used to simulate commands that have no explicit behavior specified using *`set_command()`*.

Parameters

- **stdout** – A string representing the simulated content written by the process to the stdout pipe.
- **stderr** – A string representing the simulated content written by the process to the stderr pipe.
- **returncode** – An integer representing the return code of the simulated process.
- **pid** – An integer representing the process identifier of the simulated process. This is useful if you have code the prints out the pids of running processes.
- **poll_count** – Specifies the number of times *MockPopen.poll()* can be called before `MockPopen.returncode` is set and returned by *MockPopen.poll()*.

terminate()
Simulate calls to `subprocess.Popen.terminate()`

wait()
Simulate calls to `subprocess.Popen.wait()`

For details of how to install the package or get involved in its development, please see the sections below:

CHAPTER 14

Installation Instructions

If you want to experiment with TestFixtures, the easiest way to install it is to do the following in a virtualenv:

```
pip install testfixtures
```

If your package uses setuptools and you decide to use TestFixtures, then you should do one of the following:

- Specify `testfixtures` in the `tests_require` parameter of your package's call to `setup` in `setup.py`.
- Add an `extra_requires` parameter in your call to `setup` as follows:

```
setup(
    # other stuff here
    extras_require=dict(
        test=['testfixtures'],
        )
    )
```

CHAPTER 15

Development

This package is developed using continuous integration which can be found here:

https://travis-ci.org/Simplistix/testfixtures

The latest development version of the documentation can be found here:

http://testfixtures.readthedocs.org/en/latest/

If you wish to contribute to this project, then you should fork the repository found here:

https://github.com/Simplistix/testfixtures/

Once that has been done and you have a checkout, you can follow these instructions to perform various development tasks:

Setting up a virtualenv

The recommended way to set up a development environment is to turn your checkout into a virtualenv and then install the package in editable form as follows:

```
$ virtualenv .
$ bin/pip install -U -e .[test,build]
```

Running the tests

Once you've set up a virtualenv, the tests can be run as follows:

```
$ source bin/activate
$ pytest
```

Building the documentation

The Sphinx documentation is built by doing the following from the directory containing setup.py:

```
$ source bin/activate
$ cd docs
$ make html
```

To check that the description that will be used on PyPI renders properly, do the following:

```
$ python setup.py --long-description | rst2html.py > desc.html
```

The resulting `desc.html` should be checked by opening in a browser.

Making a release

To make a release, just update `versions.txt`, update the change log, tag it and push to https://github.com/Simplistix/testfixtures and Travis CI should take care of the rest.

Once Travis CI is done, make sure to go to https://readthedocs.org/projects/testfixtures/versions/ and make sure the new release is marked as an Active Version.

CHAPTER 16

Changes

5.1.1 (8 June 2017)

- Fix support for Django 1.9 in `testfixtures.django.compare_model()`.

5.1.0 (8 June 2017)

- Added support for including non-edit able fields to the `comparer` used by *`compare()`* when comparing *django* `Model` instances.

5.0.0 (5 June 2017)

- Move from nose to pytest for running tests.
- Switch from manuel to sybil for checking examples in documentation. This introduces a backwards incompatible change in that `FileParser` replaces the Manuel plugin that is no longer included.
- Add a 'tick' method to *`test_datetime`*, *`test_date`* and *`test_time`*, to advance the returned point in time, which is particularly helpful when `delta` is set to zero.

4.14.3 (15 May 2017)

- Fix build environment bug in `.travis.yml` that caused bad tarballs.

4.14.2 (15 May 2017)

- New release as it looks like Travis mis-built the 4.14.1 tarball.

4.14.1 (15 May 2017)

- Fix mis-merge.

4.14.0 (15 May 2017)

- Added helpers for testing with *django* `Model` instances.

4.13.5 (1 March 2017)

- *compare()* now correctly compares nested empty dictionaries when using `ignore_eq=True`.

4.13.4 (6 February 2017)

- Keep the Reproducible Builds guys happy.

4.13.3 (13 December 2016)

- *compare()* now better handles equality comparison with `ignore_eq=True` when either of the objects being compared cannot be hashed.

4.13.2 (16 November 2016)

- Fixed a bug where a *LogCapture* wouldn't be cleared when used via *log_capture()* on a base class and sub class execute the same test.

Thanks to "mlabonte" for the bug report.

4.13.1 (2 November 2016)

- When `ignore_eq` is used with *compare()*, fall back to comparing by hash if not type-specific comparer can be found.

4.13.0 (2 November 2016)

- Add support to *compare()* for ignoring broken `__eq__` implementations.

4.12.0 (18 October 2016)

- Add support for specifying a callable to extract rows from log records when using *LogCapture*.
- Add support for recursive comparison of log messages with *LogCapture*.

4.11.0 (12 October 2016)

- Allow the attributes returned in `LogCapture.actual()` rows to be specified.
- Allow a default to be specified for encoding in *TempDirectory.read()* and *TempDirectory.write()*.

4.10.1 (5 September 2016)

- Better docs for *TempDirectory.compare()*.
- Remove the need for expected paths supplied to *TempDirectory.compare()* to be in sorted order.
- Document a good way of restoring `stdout` when in a debugger.
- Fix handling of trailing slashes in *TempDirectory.compare()*.

Thanks to Maximilian Albert for the *TempDirectory.compare()* docs.

4.10.0 (17 May 2016)

- Fixed examples in documentation broken in 4.5.1.
- Add *RangeComparison* for comparing against values that fall in a range.
- Add *set_default()* to *MockPopen*.

Thanks to Asaf Peleg for the *RangeComparison* implementation.

4.9.1 (19 February 2016)

- Fix for use with PyPy, broken since 4.8.0.

Thanks to Nicola Iarocci for the pull request to fix.

4.9.0 (18 February 2016)

- Added the *suffix* parameter to *compare()* to allow failure messages to include some additional context.
- Update package metadata to indicate Python 3.5 compatibility.

Thanks for Felix Yan for the metadata patch.

Thanks to Wim Glenn for the suffix patch.

4.8.0 (2 February 2016)

- Introduce a new *Replace* context manager and make *Replacer* callable. This gives more succinct and easy to read mocking code.
- Add *ShouldWarn* and *ShouldNotWarn* context managers.

4.7.0 (10 December 2015)

- Add the ability to pass `raises=False` to *compare()* to just get the resulting message back rather than having an exception raised.

4.6.0 (3 December 2015)

- Fix a bug that mean symlinked directories would never show up when using *TempDirectory.compare()* and friends.
- Add the `followlinks` parameter to *TempDirectory.compare()* to indicate that symlinked or hard linked directories should be recursed into when using `recursive=True`.

4.5.1 (23 November 2015)

- Switch from `cStringIO` to `StringIO` in *OutputCapture* to better handle unicode being written to *stdout* or *stderr*.

Thanks to "tell-k" for the patch.

4.5.0 (13 November 2015)

- *LogCapture*, *OutputCapture* and *TempDirectory* now explicitly show what is expected versus actual when reporting differences.

Thanks to Daniel Fortunov for the pull request.

4.4.0 (1 November 2015)

- Add support for labelling the arguments passed to *compare()*.
- Allow `expected` and `actual` keyword parameters to be passed to *compare()*.
- Fix `TypeError: unorderable types` when *compare()* found multiple differences in sets and dictionaries on Python 3.
- Add official support for Python 3.5.
- Drop official support for Python 2.6.

Thanks to Daniel Fortunov for the initial ideas for explicit `expected` and `actual` support in *compare()*.

4.3.3 (15 September 2015)

- Add wheel distribution to release.
- Attempt to fix up various niggles from the move to Travis CI for doing releases.

4.3.2 (15 September 2015)

- Fix broken 4.3.1 tag.

4.3.1 (15 September 2015)

- Fix build problems introduced by moving the build process to Travis CI.

4.3.0 (15 September 2015)

- Add *TempDirectory.compare()* with a cleaner, more explicit API that allows comparison of only the files in a temporary directory.
- Deprecate *TempDirectory.check()*, *TempDirectory.check_dir()* and *TempDirectory.check_all()*
- Relax absolute-path rules so that if it's inside the *TempDirectory*, it's allowed.
- Allow *OutputCapture* to separately check output to `stdout` and `stderr`.

4.2.0 (11 August 2015)

- Add *MockPopen*, a mock helpful when testing code that uses `subprocess.Popen`.
- *ShouldRaise* now subclasses `object`, so that subclasses of it may use `super()`.
- Drop official support for Python 3.2.

Thanks to BATS Global Markets for donating the code for *MockPopen*.

4.1.2 (30 January 2015)

- Clarify documentation for `name` parameter to *LogCapture*.
- *ShouldRaise* now shows different output when two exceptions have the same representation but still differ.
- Fix bug that could result in a `dict` comparing equal to a `list`.

Thanks to Daniel Fortunov for the documentation clarification.

4.1.1 (30 October 2014)

- Fix bug that prevented logger propagation to be controlled by the *log_capture* decorator.

Thanks to John Kristensen for the fix.

4.1.0 (14 October 2014)

- Fix *compare()* bug when dict instances with tuple keys were not equal.
- Allow logger propagation to be controlled by *LogCapture*.
- Enabled disabled loggers if a *LogCapture* is attached to them.

Thanks to Daniel Fortunov for the *compare()* fix.

4.0.2 (10 September 2014)

- Fix "maximum recursion depth exceeded" when comparing a string with bytes that did not contain the same character.

4.0.1 (4 August 2014)

- Fix bugs when string compared equal and options to *compare()* were used.
- Fix bug when strictly comparing two nested structures containing identical objects.

4.0.0 (22 July 2014)

- Moved from buildout to virtualenv for development.
- The identity singleton is no longer needed and has been removed.
- *compare()* will now work recursively on data structures for which it has registered comparers, giving more detailed feedback on nested data structures. Strict comparison will also be applied recursively.
- Re-work the interfaces for using custom comparers with *compare()*.
- Better feedback when comparing collections.namedtuple() instances.
- Official support for Python 3.4.

Thanks to Yevgen Kovalienia for the typo fix in *Mocking dates and times*.

3.1.0 (25 May 2014)

- Added *RoundComparison* helper for comparing numerics to a specific precision.
- Added unless parameter to *ShouldRaise* to cover some very specific edge cases.

- Fix missing imports that showed up `TempDirectory` had to do the "convoluted folder delete" dance on Windows.

Thanks to Jon Thompson for the `RoundComparison` implementation.

Thanks to Matthias Lehmann for the import error reports.

3.0.2 (7 April 2014)

- Document `ShouldRaise.raised` and make it part of the official API.
- Fix rare failures when cleaning up `TempDirectory` instances on Windows.

3.0.1 (10 June 2013)

- Some documentation tweaks and clarifications.
- Fixed a bug which masked exceptions when using `compare()` with a broken generator.
- Fixed a bug when comparing a generator with a non-generator.
- Ensure `LogCapture` cleans up global state it may effect.
- Fixed replacement of static methods using a `Replacer`.

3.0.0 (5 March 2013)

- Added compatibility with Python 3.2 and 3.3.
- Dropped compatibility with Python 2.5.
- Removed support for the following obscure uses of `should_raise`:

```
should_raise(x, IndexError)[1]
should_raise(x, KeyError)['x']
```

- Dropped the *mode* parameter to `TempDirectory.read()`.
- `TempDirectory.makedir()` and `TempDirectory.write()` no longer accept a *path* parameter.
- `TempDirectory.read()` and `TempDirectory.write()` now accept an *encoding* parameter to control how non-byte data is decoded and encoded respectively.
- Added the *prefix* parameter to `compare()` to allow failure messages to be made more informative.
- Fixed a problem when using sub-second deltas with `test_time()`.

2.3.5 (13 August 2012)

- Fixed a bug in `compare_dict()` that mean the list of keys that were the same was returned in an unsorted order.

2.3.4 (31 January 2012)

- Fixed compatibility with Python 2.5
- Fixed compatibility with Python 2.7
- Development model moved to continuous integration using Jenkins.
- Introduced Tox based testing to ensure packaging and dependencies are as expected.
- 100% line and branch coverage with tests.
- Mark `test_datetime`, `test_date` and `test_time` such that nose doesn't mistake them as tests.

2.3.3 (12 December 2011)

- Fixed a bug where when a target was replaced more than once using a single `Replacer`, `restore()` would not correctly restore the original.

2.3.2 (10 November 2011)

- Fixed a bug where attributes and keys could not be removed by a `Replacer` as described in *Removing attributes and dictionary items* if the attribute or key might not be there, such as where a test wants to ensure an `os.environ` variable is not set.

2.3.1 (8 November 2011)

- Move to use nose for running the TestFixtures unit tests.
- Fixed a bug where `tdatetime.now()` returned an instance of the wrong type when *tzinfo* was passed in *strict mode*.

2.3.0 (11 October 2011)

- `Replacer`, `TempDirectory`, `LogCapture` and `TestComponents` instances will now warn if the process they are created in exits without them being cleaned up. Instances of these classes should be cleaned up at the end of each test and these warnings serve to point to a cause for possible mysterious failures elsewhere.

2.2.0 (4 October 2011)

- Add a *strict mode* to `test_datetime` and `test_date`. When used, instances returned from the mocks are instances of those mocks. The default behaviour is now to return instances of the real `datetime` and `date` classes instead, which is usually much more useful.

2.1.0 (29 September 2011)

- Add a *strict mode* to `compare()`. When used, it ensures that the values compared are not only equal but also of the same type. This mode is not used by default, and the default mode restores the more commonly useful functionality where values of similar types but that aren't equal give useful feedback about differences.

2.0.1 (23 September 2011)

- add back functionality to allow comparison of generators with non-generators.

2.0.0 (23 September 2011)

- `compare()` now uses a registry of comparers that can be modified either by passing a *registry* option to `compare()` or, globally, using the `register()` function.
- added a comparer for `set` instances to `compare()`.
- added a new *show_whitespace* parameter to `compare_text()`, the comparer used when comparing strings and unicodes with `compare()`.
- The internal queue for `test_datetime` is now considered to be in local time. This has implication on the values returned from both `now()` and `utcnow()` when *tzinfo* is passed to the `test_datetime` constructor.
- `set()` and `add()` on `test_date`, `test_datetime` and `test_time` now accept instances of the appropriate type as an alternative to just passing in the parameters to create the instance.
- Refactored the monolithic `__init__.py` into modules for each type of functionality.

1.12.0 (16 August 2011)

- Add a `captured` property to `OutputCapture` so that more complex assertion can be made about the output that has been captured.
- `OutputCapture` context managers can now be temporarily disabled using their `disable()` method.
- Logging can now be captured only when it exceeds a specified logging level.
- The handling of timezones has been reworked in both `test_datetime()` and `test_time()`. This is not backwards compatible but is much more useful and correct.

1.11.3 (3 August 2011)

- Fix bugs where various `test_date()`, `test_datetime()` and `test_time()` methods didn't accept keyword parameters.

1.11.2 (28 July 2011)

- Fix for 1.10 and 1.11 releases that didn't include non-.py files as a result of the move from subversion to git.

1.11.1 (28 July 2011)

- Fix bug where *tdatetime.now()* didn't accept the *tz* parameter that datetime.datetime.now() did.

1.11.0 (27 July 2011)

- Give more useful output when comparing dicts and their subclasses.
- Turn *should_raise* into a decorator form of *ShouldRaise* rather than the rather out-moded wrapper function that it was.

1.10.0 (19 July 2011)

- Remove dependency on zope.dottedname.
- Implement the ability to mock out dict and list items using *Replacer* and *replace()*.
- Implement the ability to remove attributes and dict items using *Replacer* and *replace()*.

1.9.2 (20 April 2011)

- Fix for issue #328: *utcnow()* of *test_datetime()* now returns items from the internal queue in the same way as *now()*.

1.9.1 (11 March 2011)

- Fix bug when *ShouldRaise* context managers incorrectly reported what exception was incorrectly raised when the incorrectly raised exception was a KeyError.

1.9.0 (11 February 2011)

- Added TestComponents for getting a sterile registry when testing code that uses zope.component.

1.8.0 (14 January 2011)

- Added full Sphinx-based documentation.
- added a Manuel plugin for reading and writing files into a *TempDirectory*.
- any existing log handlers present when a *LogCapture* is installed for a particular logger are now removed.
- fix the semantics of *should_raise*, which should always expect an exception to be raised!
- added the *ShouldRaise* context manager.
- added recursive support to *TempDirectory.listdir()* and added the new *TempDirectory.check_all()* method.

- added support for forward-slash separated paths to all relevant *TempDirectory* methods.
- added *TempDirectory.getpath()* method.
- allow files and directories to be ignored by a regular expression specification when using *TempDirectory*.
- made *Comparison* objects work when the attributes expected might be class attributes.
- re-implement *test_time()* so that it uses the correct way to get timezone-less time.
- added *set()* along with *delta* and *delta_type* parameters to *test_date()*, *test_datetime()* and *test_time()*.
- allow the date class returned by the *tdatetime.date()* method to be configured.
- added the *OutputCapture* context manager.
- added the *StringComparison* class.
- added options to ignore trailing whitespace and blank lines when comparing multi-line strings with *compare()*.
- fixed bugs in the handling of some exception types when using *Comparison*, *ShouldRaise* or *should_raise*.
- changed *wrap()* to correctly set __name__, along with some other attributes, which should help when using the decorators with certain testing frameworks.

1.7.0 (20 January 2010)

- fixed a bug where the @replace decorator passed a classmethod rather than the replacment to the decorated callable when replacing a classmethod
- added set method to test_date, test_datetime and test_time to allow setting the parameters for the next instance to be returned.
- added delta and delta_type parameters to test_date,test_datetime and test_time to control the intervals between returned instances.

1.6.2 (23 September 2009)

- changed Comparison to use __eq__ and __ne__ instead of the deprecated __cmp__
- documented that order matters when using Comparisons with objects that implement __eq__ themselves, such as instances of Django models.

1.6.1 (06 September 2009)

- @replace and Replacer.replace can now replace attributes that may not be present, provided the *strict* parameter is passed as False.
- should_raise now catches BaseException rather than Exception so raising of SystemExit and KeyboardInterrupt can be tested.

1.6.0 (09 May 2009)

- added support for using TempDirectory, Replacer and LogCapture as context managers.
- fixed test failure in Python 2.6.

1.5.4 (11 Feb 2009)

- fix bug where should_raise didn't complain when no exception was raised but one was expected.
- clarified that the return of a should_raise call will be None in the event that an exception is raised but no expected exception is specified.

1.5.3 (17 Dec 2008)

- should_raise now supports methods other than __call__

1.5.2 (14 Dec 2008)

- added *makedir* and *check_dir* methods to TempDirectory and added support for sub directories to *read* and *write*

1.5.1 (12 Dec 2008)

- added *path* parameter to *write* method of TempDirectory so that the full path of the file written can be easilly obtained

1.5.0 (12 Dec 2008)

- added handy *read* and *write* methods to TempDirectory for creating and reading files in the temporary directory
- added support for rich comparison of objects that don't support vars()

1.4.0 (12 Dec 2008)

- improved representation of failed Comparison
- improved representation of failed compare with sequences

1.3.1 (10 Dec 2008)

- fixed bug that occurs when directory was deleted by a test that use tempdir or TempDirectory

1.3.0 (9 Dec 2008)

- added TempDirectory helper
- added tempdir decorator

1.2.0 (3 Dec 2008)

- LogCaptures now auto-install on creation unless configured otherwise
- LogCaptures now have a clear method
- LogCaptures now have a class method uninstall_all that uninstalls all instances of LogCapture. Handy for a tearDown method in doctests.

1.1.0 (3 Dec 2008)

- add support to Comparisons for only comparing some attributes
- move to use zope.dottedname

1.0.0 (26 Nov 2008)

- Initial Release

CHAPTER 17

License

```
Copyright (c) 2008-2015 Simplistix Ltd
Copyright (c) 2015-2017 Chris Withers

Permission is hereby granted, free of charge, to any person
obtaining a copy of this software and associated documentation
files (the "Software"), to deal in the Software without restriction,
including without limitation the rights to use, copy, modify, merge,
publish, distribute, sublicense, and/or sell copies of the Software,
and to permit persons to whom the Software is furnished to do so,
subject to the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

CHAPTER 18

Indices and tables

- genindex
- modindex
- search

Index

A

add() (testfixtures.tdate method), 83
add() (testfixtures.tdatetime method), 84
add() (testfixtures.ttime method), 85

C

captured (testfixtures.OutputCapture attribute), 76
check() (testfixtures.LogCapture method), 76
check() (testfixtures.TempDirectory method), 79
check_all() (testfixtures.TempDirectory method), 79
check_dir() (testfixtures.TempDirectory method), 79
cleanup() (testfixtures.TempDirectory method), 79
cleanup_all() (testfixtures.TempDirectory class method), 79
clear() (testfixtures.LogCapture method), 76
communicate() (testfixtures.popen.MockPopen method), 86
compare() (in module testfixtures), 81
compare() (testfixtures.OutputCapture method), 76
compare() (testfixtures.TempDirectory method), 79
compare_dict() (in module testfixtures.comparison), 82
compare_generator() (in module testfixtures.comparison), 82
compare_sequence() (in module testfixtures.comparison), 82
compare_set() (in module testfixtures.comparison), 82
compare_simple() (in module testfixtures.comparison), 82
compare_text() (in module testfixtures.comparison), 82
compare_tuple() (in module testfixtures.comparison), 82
compare_with_type() (in module testfixtures.comparison), 82
Comparison (class in testfixtures), 75
create() (testfixtures.TempDirectory method), 80

D

date() (testfixtures.tdatetime class method), 85
diff() (in module testfixtures), 82
disable() (testfixtures.OutputCapture method), 76

E

enable() (testfixtures.OutputCapture method), 76

G

generator() (in module testfixtures), 82
getpath() (testfixtures.TempDirectory method), 80

I

install() (testfixtures.LogCapture method), 76

K

kill() (testfixtures.popen.MockPopen method), 86

L

listdir() (testfixtures.TempDirectory method), 80
log_capture() (in module testfixtures), 82
LogCapture (class in testfixtures), 75

M

makedir() (testfixtures.TempDirectory method), 80
MockPopen (class in testfixtures.popen), 86

N

not_there (in module testfixtures), 86
now() (testfixtures.tdatetime class method), 85

O

OutputCapture (class in testfixtures), 76

P

path (testfixtures.TempDirectory attribute), 81
poll() (testfixtures.popen.MockPopen method), 86

R

raised (testfixtures.ShouldRaise attribute), 78
RangeComparison (class in testfixtures), 77
read() (testfixtures.TempDirectory method), 81
register() (in module testfixtures.comparison), 82

Replace (class in testfixtures), 76
replace() (in module testfixtures), 77
replace() (testfixtures.Replacer method), 77
Replacer (class in testfixtures), 77
restore() (testfixtures.Replacer method), 77
RoundComparison (class in testfixtures), 77

S

send_signal() (testfixtures.popen.MockPopen method), 86
set() (testfixtures.tdate method), 83
set() (testfixtures.tdatetime method), 85
set() (testfixtures.ttime method), 86
set_command() (testfixtures.popen.MockPopen method), 86
set_default() (testfixtures.popen.MockPopen method), 86
should_raise (class in testfixtures), 83
ShouldNotWarn (class in testfixtures), 78
ShouldRaise (class in testfixtures), 78
ShouldWarn (class in testfixtures), 78
StringComparison (class in testfixtures), 78

T

tempdir() (in module testfixtures), 83
TempDirectory (class in testfixtures), 78
terminate() (testfixtures.popen.MockPopen method), 87
test_date() (in module testfixtures), 83
test_datetime() (in module testfixtures), 84
test_time() (in module testfixtures), 85
tick() (testfixtures.tdate method), 84
tick() (testfixtures.tdatetime method), 85
tick() (testfixtures.ttime method), 86
today() (testfixtures.tdate class method), 84

U

uninstall() (testfixtures.LogCapture method), 76
uninstall_all() (testfixtures.LogCapture class method), 76
utcnow() (testfixtures.tdatetime class method), 85

W

wait() (testfixtures.popen.MockPopen method), 87
wrap() (in module testfixtures), 86
write() (testfixtures.TempDirectory method), 81