tl.testing Documentation

Release

December 17, 2011

CONTENTS

1 The tl.testing distribution **1**

1.1 Sandboxes of directories and files 1
1.2 Installing callable scripts 1
1.3 Doc-testing the graphical content of cairo surfaces 2
1.4 Working with threads in test code 2
1.5 Constructing test suites that use manuel 3

2 Narrative documentation **5**

3 API reference **7**

4 About tl.testing **9**

5 Change log for tl.testing **11**

5.1 0.5 (unreleased) 11
5.2 0.4 (2009-09-14) 11
5.3 0.3 (2009-07-23) 11
5.4 0.2.2 (2009-06-24) 12
5.5 0.2.1 (2009-06-09) 12
5.6 0.2 (2009-05-15) 12
5.7 0.1 (2008-11-11) 12

CHAPTER

ONE

THE TL.TESTING DISTRIBUTION

This package provides various utilities that can be used when writing tests.

1.1 Sandboxes of directories and files

When testing code that modifies directories and files, it is useful to be able to create and inspect a sample tree of directories and files easily. The `tl.testing.fs` module provides support for creating a tree from a textual description, listing it in the same format and clean up after itself.

In a doc test, these facilities might be used like this to create and list a directory, a file and a symbolic link:

```
>>> from tl.testing.fs import new_sandbox, ls
>>> new_sandbox("""\
... d foo
... f foo/bar asdf
... l baz -> foo/bar
... """)

>>> ls()
l baz -> foo/bar
d foo
f foo/bar asdf
```

See the file `fs.txt` found with the source code for further advice, including how to set up and tear down tests using file-system sandboxes.

1.2 Installing callable scripts

Some functionality one might want to test makes use of external programs such as a pager or a text editor. The `tl.testing.script` module provides utilities that install simple mock scripts in places where the code to be tested will find them. They take a string of Python code and create a wrapper script that sets the Python path to match that of the test and runs the code.

This is how such a mock script might be used in a doc test:

```
>>> from tl.testing.script import install
>>> script_path = install("print 'A simple script.'")
>>> print open(script_path).read()
#!...

import sys
```

```
sys.path[:] = [...]

print 'A simple script.'
```

```
>>> import subprocess
>>> sub = subprocess.Popen(script_path, shell=True, stdout=subprocess.PIPE)
>>> stdout, stderr = sub.communicate()
>>> print stdout
A simple script.
```

See the file `script.txt` found with the source code for further possibilities how to install and access mock scripts as well as how to tear down tests using mock scripts.

1.3 Doc-testing the graphical content of cairo surfaces

While it is straight-forward to compare the content of two cairo surfaces in Python code, handling graphics is beyond doc tests. However, the manuel package can be used to extract more general test cases from a text document while allowing to mix them with doc tests in a natural way.

The `tl.testing.cairo` module provides a test suite factory that uses manuel to execute graphical tests formulated as restructured-text figures. The caption of such a figure is supposed to be a literal Python expression whose value is a cairo surface, and its image is used as the test expectation.

This is how a surface might be compared to an expected image in a doc test:

```
>>> import cairo
>>> from pkg_resources import resource_filename

>>> image = resource_filename('tl.testing', 'testimages/correct.png')
```

```
.. figure:: tl/testing/testimages/correct.png

   ``cairo.ImageSurface.create_from_png(image)``
```

See the file `cairo.txt` found with the source code for further advice and documentation of the possible test output.

1.4 Working with threads in test code

The standard `TestCase` class doesn't collect errors and failures that occurred in other threads than the main one. The `tl.testing.thread` module provides thread classes and a `ThreadAwareTestCase` class to allow just that, as well as some other conveniences for tests that deal with threads: preventing expected unhandled exceptions in threads from being printed with the test output, reporting threads left behind by a test, running code in a daemon thread, joining threads and counting the threads started during the test's run time:

```
>>> import time
>>> import tl.testing.thread
>>> import unittest

>>> class SampleTest(tl.testing.thread.ThreadAwareTestCase):
...
...     def test_error_in_thread_should_be_reported(self):
...         t = self.run_in_thread(lambda: 1/0)
...         self.join(t)
...
```

```
...     def test_active_count_should_count_only_new_threads(self):
...         t = self.run_in_thread(lambda: time.sleep(0.1))
...         self.assertEqual(1, self.active_count())
...         self.join(t)
...         self.assertEqual(0, self.active_count())

>>> run(unittest.makeSuite(SampleTest))
<SET UP>
Error in test test_error_in_thread_should_be_reported (__builtin__.SampleTest)
Traceback (most recent call last):
  ...
ZeroDivisionError: integer division or modulo by zero
  Ran 2 tests with 0 failures and 1 errors in 0.117 seconds.
<TEAR DOWN>
```

1.5 Constructing test suites that use manuel

As `manuel` provides some powerful features in addition to standard doctests, manuel test suites are set up slightly differently from standard ones. The `tl.testing.doctest` module implements a `DocFileSuite` factory that can be used like the standard one but creates a test suite using manuel and allows some additional configuration related to manuel, among them the ability to interpret footnotes that used to be done using the deprecated `zope.testing.doctest`:

```
>>> sample_txt = write('sample.txt', """\
... [#footnote]_
... >>> x
... 1
...
... .. [#footnote]
...     >>> x = 1
... """)

>>> from tl.testing.doctest import DocFileSuite
>>> run(DocFileSuite(sample_txt, footnotes=True))
<SET UP>
  Ran 2 tests with 0 failures and 0 errors in 0.000 seconds.
<TEAR DOWN>

>>> sample_txt = write('sample.txt', """\
... .. code-block:: python
...     x = 1
...
... >>> x
... 1
... """)

>>> import manuel.codeblock
>>> run(DocFileSuite(sample_txt, manuel=manuel.codeblock.Manuel()))
<SET UP>
  Ran 2 tests with 0 failures and 0 errors in 0.000 seconds.
<TEAR DOWN>
```

CHAPTER

TWO

NARRATIVE DOCUMENTATION

CHAPTER

THREE

API REFERENCE

CHAPTER

FOUR

ABOUT TL.TESTING

Author Thomas Lotze <thomas@thomas-lotze.de>

Online documentation http://packages.python.org/tl.testing/

PyPI page http://pypi.python.org/pypi/tl.testing/

Issue tracker https://bitbucket.org/tlotze/tl.testing/issues/

Source code https://bitbucket.org/tlotze/tl.testing/

Current change log https://bitbucket.org/tlotze/tl.testing/raw/tip/CHANGES.txt

Support the project

CHAPTER

FIVE

CHANGE LOG FOR TL.TESTING

5.1 0.5 (unreleased)

- Added `tl.testing.thread` providing thread classes that handle or report exceptions as well as a thread-aware test case class that provides reporting on threads left behind by tests and some conveniences for using threads in test code.
- Dropped compatibility with Python 2.5 by introducing a context manager for joining threads.
- Added `tl.testing.doctest` providing a `DocFileSuite` factory which is compatible to that of standard doctest but uses pypi:*manuel* and allows for some manuel-related additional configuration. Always require **:pypi:‘manuel‘**, removed the `cairo` extra requirements.
- Fixed **:bbissue:‘5‘**: Shield test runs that are the subject of this package’s tests from options passed to runs of this package’s test suite itself.
- Updated code and development environment to newer versions of packages: require **:pypi:‘zope.testrunner‘** for tests, no longer use deprecated `zope.testing.doctest`, build with more recent **:pypi:‘tl.buildout_gtk‘**.
- Keep buildout.cfg under version control instead of base.cfg, expect overriding to happen in local.cfg.
- Updated to using **:pypi:‘Sphinx (1.1)‘** for building the documentation, require graphviz to create class inheritance diagrams, require **:pypi:‘sphinxcontrib-bitbucket‘** to generate links to bitbucket issues and **:pypi:‘sphinxcontrib-cheeseshop‘** to generate links to packages’ PyPI pages. Use a global table of contents in the sidebar.

5.2 0.4 (2009-09-14)

Cairo testing:

- Updated to using manuel 1.0.0b3, restoring Python 2.5 compatibility.
- Allowed for text in addition to the expression in figure captions, as well as multi-line captions.
- Interpret an option for excluding one or more rectangular areas of a surface from comparison with the expected image.

5.3 0.3 (2009-07-23)

Cairo testing:

- Updated to using manuel 1.0.0b2, implying the requirement of Python 2.6 and a refactoring of the DocFileSuite that loses encoding support.
- Improved the behaviour in the case that a result cannot be written to an image file.
- Write the test result to an image file also in the case that the expected image cannot be loaded.
- Disallowed absolute image paths as they would only work on Unix-like systems because of the requirement to use / as the path separator.
- Fixed DocFileSuite to accept multiple test file names, renamed the `manuel_object` parameter to just `manuel`, only include the doc test, footnote and cairo Manuels by default.
- Require a cairo.ImageSurface instead of a Surface as the value of an example's Python expression.
- Report ImageSurface format mismatches, avoid false mismatches with FORMAT_RGB24 surfaces.

Organisation:

- Moved the project to the Mercurial version control system.
- Build documentation using Sphinx.
- Improved documentation, in particular added doc strings to the code.

5.4 0.2.2 (2009-06-24)

- Updated to using the manuel 1.0.0b1 API.
- Made the cairo test runner recognize the CAIRO_TEST_RESULTS environment variable as a place to dump failed test results as png images.

5.5 0.2.1 (2009-06-09)

- Use new `manuel.codeblock` functionality in cairo image doc-testing.
- Simplified the cairo testing code a little due to a bug fix in manuel.

5.6 0.2 (2009-05-15)

- Added a manuel test runner that compares cairo surfaces to images included as ReST figures in doc test files.

5.7 0.1 (2008-11-11)

initial release

- *genindex*
- *search*