JULIEN DANJOU

# THE HACKER'S GUIDE TO PYTHON

# Contents

# 6 Unit testing

**Breaking news!** It's 2015 and there are still people who don't have a policy of testing their projects. Now, the purpose of this book is not to convince you to jump in and start unit testing. If you need to be convinced, I suggest you start by reading about the benefits of test-driven development. Writing code that is not tested is essentially useless, as there's no way to conclusively prove that it works.

This section will cover the Python tools you can use to construct a great suite of tests. We'll talk about how you can utilise them to enhance your software, making it rock-solid and regression free!

## 6.1 The basics

Contrary to what you may believe, the writing and running of unit tests is really simple in Python. It's not intrusive or disruptive, and it's going to help you and other developers a lot in maintaining your software.

Your tests should be stored inside a `tests` submodule of your application or library. This allows you to ship the tests as part of your module, so that they can be run or reused by anyone – even once your software is installed – without necessarily using the source package. This also prevents them from being installed by mistake in a top-level `tests` module.

It's usually simpler to use a hierarchy in your test tree that mimics the hierarchy you have in your module tree. This means that the tests covering the code of `mylib/foobar.py` should be inside `mylib/tests/test_foobar.py`; this makes things simpler when looking for the tests relating to a particular file.

**Example 6.1** A really simple test in `test_true.py`

```
def test_true():
    assert True
```

This is the most simple unit test that can be written. To run it, you simply need to load the `test_true.py` file and run the `test_true` function defined within.

Obviously, following these steps for all of your test files and functions would be a pain. This is where the *nose* package comes to the rescue – once installed, it provides the `nosetests` command, which loads every file whose name starts with `test_` and then executes all functions within that start with `test_`.

Therefore, with the `test_true.py` file in our source tree, running `nosetests` will give us the following output:

```
$ nosetests -v
test_true.test_true ... ok


----------------------------------------------------------------
Ran 1 test in 0.003s


OK
```

On the other hand, as soon as a test fails, the output changes to indicate the failure, accompanied by the whole traceback.

```
% nosetests -v
test_true.test_true ... ok
test_true.test_false ... FAIL
```

```
======================================================
FAIL: test_true.test_false

Traceback (most recent call last):
  File "/usr/lib/python2.7/dist-packages/nose/case.py", line 197, in ↩
     runTest
    self.test(*self.arg)
  File "/home/jd/test_true.py", line 5, in test_false
    assert False
AssertionError
----------------------------------------------------------
Ran 2 tests in 0.003s

FAILED (failures=1)
```

A test fails as soon as an `AssertionError` exception is raised; `assert` does indeed raise an `AssertionError` as soon as its argument is evaluated to something false (`False`, `None`, 0…). If any other exception is raised, the test also errors out.

Simple, isn't it? While simplistic, this approach is used by a lot of small projects, and works very well. They don't require tools or libraries other than than `nose`, and relying on `assert` is good enough.

However, as you start to write more sophisticated tests, you'll start to become frustrated by things like the use of `assert`. Consider the following test:

```
def test_key():
    a = ['a', 'b']
    b = ['b']
    assert a == b
```

When running `nosetests`, it gives the following output:

```
$ nosetests -v
test_complicated.test_key ... FAIL

======================================================================
FAIL: test_complicated.test_key
Traceback (most recent call last):
  File "/usr/lib/python2.7/dist-packages/nose/case.py", line 197, in  ↩
     runTest
    self.test(*self.arg)
  File "/home/jd/test_complicated.py", line 4, in test_key
    assert a == b
AssertionError

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
```

Alright, so `a` and `b` are different and this test doesn't pass. But how are they different? `assert` doesn't give us this information, just states that the assertion is wrong – not particularly useful.

Also, with such a basic zero framework approach, advanced usage such as skipping tests or executing actions before or after running every test can become painful.

This is where the `unittest` package comes in handy. It provides tools that will help covering all of that – and good news is that `unittest` is part of the Python standard library.

**Warning**

`unittest` has been largely improved starting with Python 2.7, so if you are supporting earlier versions of Python you may want to use its backport named `unittest2`. If you need to support Python 2.6, you can then use the following snippet to import the correct module for any Python versions at runtime:

```
try:
    import unittest2 as unittest
except ImportError:
    import unittest
```

If we rewrite the previous example using `unittest`, this is what it will look like:

```
import unittest


class TestKey(unittest.TestCase):
    def test_key(self):
        a = ['a', 'b']
        b = ['b']
        self.assertEqual(a, b)
```

As you can see, the implementation isn't much more complicated. All you have to do is create a class that inherits from `unittest.TestCase`, and write a method that runs a test. Instead of using `assert`, we rely on a method provided by `unittest.TestCase` that provides an equality tester. When run, it outputs the following:

```
$ nosetests -v
test_key (test_complicated.TestKey) ... FAIL


======================================================
FAIL: test_key (test_complicated.TestKey)
Traceback (most recent call last):
```

```
  File "/home/jd/Source/python-book/test_complicated.py", line 7, in  ↩
     test_key
    self.assertEqual(a, b)
AssertionError: Lists differ: ['a', 'b'] != ['b']

First differing element 0:
a
b

First list contains 1 additional elements.
First extra element 1:
b

- ['a', 'b']
+ ['b']

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
```

As you can see, the output is much more useful. An assertion error is still raised, and the test is still being failed, but at least we have real information about why it's failing, which can help us to fix the problem. This is why you should definitely **never** use `assert` when writing test cases. Anyone who tries to hack your code and ends up failing a test will definitely thank you for having not used `assert`, and having thereby providing him/her with debugging information right away.

`unittest` provides a few test functions that you can use to specialize your tests, such as: `assertDictEqual`, `assertEqual`, `assertTrue`, `assertFalse`, `assertGreater`, `assertGreaterEqual`, `assertIn`, `assertIs`, `assertIsIntance`, `assertIsNone`, `asser`

tIsNot, assertIsNotNone, assertItemsEqual, assertLess, assertLessEqual, assertListEqual, assertMultiLineEqual, assertNotAlmostEqual, assertNotEqual, assertTupleEqual, assertRaises, assertRaisesRegexp, assertRegexpMatches, etc. It would be a good idea to go through `pydoc unittest` and discover them all.

It's also possible to deliberately fail a test right away using the `fail(msg)` method. This can be convenient when you know that a particular part of your code will definitely raise an error if executed, but there isn't a particular assertion to check for.

**Example 6.2** Failing a test

```
import unittest


class TestFail(unittest.TestCase):
    def test_range(self):
        for x in range(5):
            if x > 4:
                self.fail("Range returned a too big value: %d" % x)
```

It's sometimes useful skip a test if it can't be run – for example, you may wish to run a test conditionally based on the presence or absence of a particular library. To that end, you can raise the **`unittest.SkipTest`** exception. When the test is raised, it is simply marked as having been skipped. The convenient method `unittest.TestCase.skipTest()` can be used rather than raising the exception manually, as can the `unittest.skip` decorator:

**Example 6.3** Skipping tests

```
import unittest


try:
    import mylib
except ImportError:
    mylib = None
```

```
class TestSkipped(unittest.TestCase):
    @unittest.skip("Do not run this")
    def test_fail(self):
        self.fail("This should not be run")

    @unittest.skipIf(mylib is None, "mylib is not available")
    def test_mylib(self):
        self.assertEqual(mylib.foobar(), 42)

    def test_skip_at_runtime(self):
        if True:
            self.skipTest("Finally I don't want to run it")
```

When executed, this test file will output the following:

```
$ python -m unittest -v test_skip
test_fail (test_skip.TestSkipped) ... skipped 'Do not run this'
test_mylib (test_skip.TestSkipped) ... skipped 'mylib is not available'
test_skip_at_runtime (test_skip.TestSkipped) ... skipped "Finally I don't ↩
   want to run it"

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK (skipped=3)
```

**Tip**

As you may have noticed in Example 6.3, the `unittest` module provides a way to execute a Python module that contains tests. It is less convenient than using `nosetests`, as it does not discover test files on its own, but it can still be useful for running a particular test module.

In many cases, there's a need to execute a set of common actions before and after running a test. `unittest` provides two particular methods called **`setUp`** and **`tearDown`** that are executed each time one of the test methods of a class is about to, or has been, called.

**Example 6.4** Using `setUp` with `unittest`

```
import unittest

class TestMe(unittest.TestCase):
    def setUp(self):
        self.list = [1, 2, 3]

    def test_length(self):
        self.list.append(4)
        self.assertEqual(len(self.list), 4)

    def test_has_one(self):
        self.assertEqual(len(self.list), 3)
        self.assertIn(1, self.list)
```

In this case, `setUp` is called before running `test_length` and before running `test_has_one`. It can be really handy to create objects that are worked with during each test; but you need to be sure that they get recreated in a clean state before each test method is called. This is really useful for creating test environments, often referred

to as ”fixtures” (see Section 6.2).

**Tip**

When using `nosetests`, you often might want to run only one particular test. You can select which test you want to run by passing it as an argument – the syntax is: *path.to.your.module:ClassOfYourTest.test_method*. Be sure that there's a colon between the module path and the class name. You can also specify *path.to.your.module:ClassOfYourTest* to execute an entire class, or *path.to.your.module* to execute an entire module.

**Tip**

It's possible to run tests in parallel to speed things up. Simply add the `--processes=N` option to your `nosetests` invocation to spawn several `nosetests` processes. However, `testrepository` is a better alternative – this is discussed in Section 6.5.

## 6.2 Fixtures

In unit testing, fixtures represent components that are set up before a test, and cleaned up after the test is done. It's usually a good idea to build a special kind of component for them, as they are reused in a lot of different places. For example, if you need an object which represents the configuration state of your application, there's a chance you may want it to be initialized before each test, and reset to its default values when the test is done. Relying on temporary file creation also requires that the file is created before the test starts, and deleted once the test is done.

`unittest` only provides the `setUp` and `tearDown` functions we already evoked. However, a mechanism exists to hook into these. The **`fixtures`** Python module (not part of the standard library) provides an easy mechanism for creating fixture classes and objects, such as the `useFixture` method.

The `fixtures` modules provides a few built-in fixtures, like `fixtures.EnvironmentVariable` – useful for adding or changing a variable in `os.environ` that will be reset upon test exit.

**Example 6.5** Using `fixtures.EnvironmentVariable`

```
import fixtures
import os

class TestEnviron(fixtures.TestWithFixtures):
    def test_environ(self):
        fixture = self.useFixture(
            fixtures.EnvironmentVariable("FOOBAR", "42"))
        self.assertEqual(os.environ.get("FOOBAR"), "42")

    def test_environ_no_fixture(self):
        self.assertEqual(os.environ.get("FOOBAR"), None)
```

When you can identify common patterns like these, it's a good idea to create a fixture that you can reuse over all your test cases. This greatly simplifies the logic, and shows exactly what you are testing and in what manner.

**Note**

If you're wondering why the base class `unittest.TestCase` isn't used in the examples in this section, it's because `fixtures.TestWithFixtures` inherits from it.

## 6.3 Mocking

Mock objects are simulated objects that mimic the behaviour of real application objects, but in particular and controlled ways. These are especially useful in creat-

ing environments that describe precisely the conditions for which you would like to test code.

If you are writing an HTTP client, it's likely impossible (or at least extremely complicated) to spawn the HTTP server and test it through all scenarios, making it return every possible value. It's especially difficult to test for all failure scenarios.

A much simpler option is to build a set of mock objects that model these particular scenarios, and to use them as environment for testing your code.

The standard library for creating mock objects in Python is `mock`. Starting with Python 3.3, it has been merged into the Python standard library as `unittest.mock`. You can therefore use a snippet like:

```
try:
    from unittest import mock
except ImportError:
    import mock
```

To maintain backward compatibility between Python 3.3 and earlier versions.

Mock is pretty simple to use:

**Example 6.6** Basic mock usage

```
>>> import mock
>>> m = mock.Mock()
>>> m.some_method.return_value = 42
>>> m.some_method()
42
>>> def print_hello():
...     print("hello world!")
...
>>> m.some_method.side_effect = print_hello
>>> m.some_method()
```

```
hello world!
>>> def print_hello():
...     print("hello world!")
...     return 43
...
>>> m.some_method.side_effect = print_hello
>>> m.some_method()
hello world!
43
>>> m.some_method.call_count
3
```

Even using just this set of features, you should be able to mimic a lot of your internal objects in order to fake various data scenarios.

Mock uses the action/assertion pattern: this means that once your test has run, you will have to check that the actions you are mocking were correctly executed.

**Example 6.7** Checking method calls

```
>>> import mock
>>> m = mock.Mock()
>>> m.some_method('foo', 'bar')
<Mock name='mock.some_method()' id='26144272'>
>>> m.some_method.assert_called_once_with('foo', 'bar')
>>> m.some_method.assert_called_once_with('foo', mock.ANY)
>>> m.some_method.assert_called_once_with('foo', 'baz')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python2.7/dist-packages/mock.py", line 846, in ↩
     assert_called_once_with
    return self.assert_called_with(*args, **kwargs)
  File "/usr/lib/python2.7/dist-packages/mock.py", line 835, in ↩
```

```
     assert_called_with
    raise AssertionError(msg)
AssertionError: Expected call: some_method('foo', 'baz')
Actual call: some_method('foo', 'bar')
```

As you can see, it's easy enough to pass a *mock* object to any part of your code, and to check later if the code has been called with whatever argument it was supposed to have. If you don't know what arguments may have been passed, you can use `mock.ANY` as a value; that will match any argument passed to your mock method.

Sometimes you may need to patch some function, method or object from an external module. `mock` provides a set of patching functions to that end.

**Example 6.8** Using `mock.patch`

```
>>> import mock
>>> import os
>>> def fake_os_unlink(path):
...     raise IOError("Testing!")
...
>>> with mock.patch('os.unlink', fake_os_unlink):
...     os.unlink('foobar')
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in fake_os_unlink
IOError: Testing!
```

With the `mock.patch` method, it's possible to change any part of an external piece of code – making it behave in the required way in order to test all conditions in your software.

**Example 6.9** Using `mock.patch` to test a set of behaviour

```
import requests
import unittest
import mock

class WhereIsPythonError(Exception):
    pass

def is_python_still_a_programming_language():
    try:
        r = requests.get("http://python.org")
    except IOError:
        pass
    else:
        if r.status_code == 200:
            return 'Python is a programming language' in r.content
    raise WhereIsPythonError("Something bad happened")

def get_fake_get(status_code, content):
    m = mock.Mock()
    m.status_code = status_code
    m.content = content
    def fake_get(url):
        return m
    return fake_get

def raise_get(url):
    raise IOError("Unable to fetch url %s" % url)

class TestPython(unittest.TestCase):
    @mock.patch('requests.get', get_fake_get(
```

```
        200, 'Python is a programming language for sure'))
    def test_python_is(self):
        self.assertTrue(is_python_still_a_programming_language())

    @mock.patch('requests.get', get_fake_get(
        200, 'Python is no more a programming language'))
    def test_python_is_not(self):
        self.assertFalse(is_python_still_a_programming_language())

    @mock.patch('requests.get', get_fake_get(
        404, 'Whatever'))
    def test_bad_status_code(self):
        self.assertRaises(WhereIsPythonError,
                          is_python_still_a_programming_language)

    @mock.patch('requests.get', raise_get)
    def test_ioerror(self):
        self.assertRaises(WhereIsPythonError,
                          is_python_still_a_programming_language)
```

Example 6.9 uses the decorator version of `mock.patch`, this does not change its behaviour, but is easier to use when you need to use mocking within the context of an entire test function.

By using mocking we can simulate any problem – such as a Web server returning a 404 error, or network issues arising. We can make sure that our code returns the correct values, or raises the correct exception in every case – ensuring that our code always behaves as expected.

## 6.4 Scenarios

When unit testing, it is common to require that a set of tests be run against different versions of an object. You may want to run the same error-handling test with a bunch of different objects that trigger that error; or you may want to run an entire test suite against different drivers.

This last case is one that we heavily relied on in *Ceilometer*. *Ceilometer* provides an abstract class that we call the storage API. Any driver can implement this base abstract class and register itself to become a driver. The software loads the configured storage driver when required, and uses the implemented storage API to store or retrieve data. In this case, what is needed is a class of unit tests that runs against each driver – meaning against each implementation of this storage API – to be sure that they conform to what the callers expect.

The natural way of doing this is to use mixin classes; on one side, you would have a class with unit tests, and on the other side a class with the specific driver usage setup.

```
import unittest


class MongoDBBaseTest(unittest.TestCase):
    def setUp(self):
        self.connection = connect_to_mongodb()


class MySQLBaseTest(unittest.TestCase):
    def setUp(self):
        self.connection = connect_to_mysql()


class TestDatabase(unittest.TestCase):
    def test_connected(self):
        self.assertTrue(self.connection.is_connected())
```

```
class TestMongoDB(TestDatabase, MongoDBBaseTest):
    pass

class TestMySQL(TestDatabase, MySQLBaseTest):
    pass
```

Unfortunately, in the long run this method is far from convenient or scalable.

A better technique does exist, using the **testscenarios** package. This package provides an easy way to run a class test against a different set of scenarios generated at run-time. Using `testscenarios`, I have rewritten part of Example 6.9 to illustrate mocking as covered in Section 6.3.

**Example 6.10** `testscenarios` basic usage

```
import mock
import requests
import testscenarios

class WhereIsPythonError(Exception):
    pass

def is_python_still_a_programming_language():
    r = requests.get("http://python.org")
    if r.status_code == 200:
        return 'Python is a programming language' in r.content
    raise WhereIsPythonError("Something bad happened")

def get_fake_get(status_code, content):
    m = mock.Mock()
    m.status_code = status_code
```

```
    m.content = content
    def fake_get(url):
        return m
    return fake_get

class TestPythonErrorCode(testscenarios.TestWithScenarios):
    scenarios = [
        ('Not found', dict(status=404)),
        ('Client error', dict(status=400)),
        ('Server error', dict(status=500)),
    ]

    def test_python_status_code_handling(self):
        with mock.patch('requests.get',
                        get_fake_get(
                            self.status,
                            'Python is a programming language for sure')):
            self.assertRaises(WhereIsPythonError,
                              is_python_still_a_programming_language)
```

Even though only one test seems to be defined, `testscenarios` runs the test three times – because we have defined three scenarios.

```
% python -m unittest -v test_scenario
test_python_status_code_handling (test_scenario.TestPythonErrorCode) ...  ↩
   ok
test_python_status_code_handling (test_scenario.TestPythonErrorCode) ...  ↩
   ok
test_python_status_code_handling (test_scenario.TestPythonErrorCode) ...  ↩
   ok
```

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

OK
```

As you can see, all we need to construct the scenario list is a tuple list that consists of the scenario name as first argument, and as a second argument the dictionary of attributes to be added to the test class for this scenario.

It is easy enough to imagine another use: where instead of storing a single value as an attribute for each test, you could instantiate a particular driver and run all the tests of the class against it.

**Example 6.11** Using `testscenarios` to test drivers

```
import testscenarios
from myapp import storage

class TestPythonErrorCode(testscenarios.TestWithScenarios):
    scenarios = [
        ('MongoDB', dict(driver=storage.MongoDBStorage())),
        ('SQL', dict(driver=storage.SQLStorage())),
        ('File', dict(driver=storage.FileStorage())),
    ]

    def test_storage(self):
        self.assertTrue(self.driver.store({'foo': 'bar'}))

    def test_fetch(self):
        self.assertEqual(self.driver.fetch('foo'), 'bar')
```

**Note**

If you wonder why there is no need to use the base class `unittest.TestCase` in the previous examples, it's because `testscenarios.TestWithScenarios` inherits from it.

## 6.5 Test streaming and parallelism

When performing a lot of tests, it can be useful to analyze them as they are run. The default behaviour of tools like `nosetests` is to output the result to *stdout* – which is not really convenient to parse or analyze.

**`subunit`** is a Python module that provides a streaming protocol for test results. It allows for a number of interesting things, such as aggregating test results [1] or to record and archive test runs, etc.

Running a test using `subunit` is simple enough:

```
$ python -m subunit.run test_scenario
```

The output of this command is binary data, so unless you have the ability to sight-read the *subunit* protocol, it wouldn't be interesting to reproduce it's output directly here. However, `subunit` also comes with a set of tools to transform this binary stream into something smoother:

**Example 6.12** Using `subunit2pyunit`

```
$ python -m subunit.run test_scenario | subunit2pyunit
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Not ↩
   found)
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Not ↩
   found) ... ok
```

[1] Even from different source programs or languages

```
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Client ↩
   error)
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Client ↩
   error) ... ok
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Server ↩
   error)
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Server ↩
   error) ... ok

----------------------------------------------------------
Ran 3 tests in 0.061s

OK
```

Now this is something that we can understand – you should recognize the test suite with scenarios from Section 6.4. Other tools worth mentioning include `subunit2csv`, `subunit2gtk` and `subunit2junitxml`.

`subunit` is also able to automatically discover which test to run, when it is passed the `discover` argument.

```
$ python -m subunit.run discover | subunit2pyunit
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Not ↩
   found)
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Not ↩
   found) ... ok
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Client ↩
   error)
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Client ↩
   error) ... ok
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Server ↩
   error)
```

```
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Server ↩
   error) ... ok


----------------------------------------------------------------
Ran 3 tests in 0.061s


OK
```

You can list tests, rather than running them, by passing the argument `--list`. To view the results, you can use `subunit-ls`:

```
$ python -m subunit.run discover --list | subunit-ls --exists
test_request.TestPython.test_bad_status_code
test_request.TestPython.test_ioerror
test_request.TestPython.test_python_is
test_request.TestPython.test_python_is_not
test_scenario.TestPythonErrorCode.test_python_status_code_handling
```

**Tip**

You can also load a list of tests that you want to run – rather than running all tests – by using the `--load-list` option.

In large applications the number of tests can be overwhelming, so having programs to handle the stream of results is very useful. The **`testrepository`** package is intended to do just that; it provides the `testr` program, which you can use to handle a database of your test run.

```
$ testr init
$ touch .testr.conf
% python -m subunit.run test_scenario | testr load
Ran 4 tests in 0.001s
```

```
PASSED (id=0)
$ testr failing
PASSED (id=0)
$ testr last
Ran 3 tests in 0.001s
PASSED (id=0)
$ testr slowest
Test id                                          Runtime (s)
-----------------------------------------------  -----------
test_python_status_code_handling(Not found)      0.000
test_python_status_code_handling(Server error)   0.000
test_python_status_code_handling(Client error)   0.000
$ testr stats
runs=1
```

Once the *subunit* stream of tests has been run and loaded inside *testrepository*, it is possible to manipulate it easily using the `testr` command.

Obviously, this is tedious to do by hand each time you want to run tests. Instead, you should teach `testr` how it should run your tests, so that it can load the results itself. This can be accomplished by editing the `.testr.conf` file at the root of your project.

**Example 6.13** A `.testr.conf` file

```
[DEFAULT]
test_command=python -m subunit.run discover . $LISTOPT $IDOPTION ❶
test_id_option=--load-list $IDFILE ❷
test_list_option=--list ❸
```

❶ Command to run when calling `testr run`

❷ Command to run to load a test list

❸ Command to run to list tests

The first line, `test_command`, is the one that is the most interesting. Now, all that we need to do to load tests into *testrepository* and perform them is to run `testr run`.

**Note**

If you're accustomed to running `nosetests`, `testr run` is now the equivalent command.

Two other options enable us to run the tests in parallel. This is simple enough to do – all you need to do is add the `--parallel` switch to `testr run`. Running your tests in parallel can speed up the process considerably.

**Example 6.14** Running `testr run --parallel`

```
$ testr run --parallel
running=python -m subunit.run discover . --list
running=python -m subunit.run discover .  --load-list /tmp/tmpiMq5Q1
running=python -m subunit.run discover .  --load-list /tmp/tmp7hYEkP
running=python -m subunit.run discover .  --load-list /tmp/tmpP_9zBc
running=python -m subunit.run discover .  --load-list /tmp/tmpTejc5J
Ran 26 (+10) tests in 0.029s (-0.001s)
PASSED (id=7, skips=3)
```

Under the hood, `testr` runs the test listing operation, splits the test list into several sublists, and creates a separate Python process to run each sublist of test. By default, the number of sublists is equal to the number of CPUs in the machine being used. You can override the number of processes that by adding the `--concurrency` flag.

```
$ testr run --parallel --concurrency=2
```

As you can imagine, there's a lot of possibilities opened up by tools such as `subunit` and `testrepository` that have only be skimmed through in this section. I believe it's worth being familiar with them, because testing can greatly influence the quality of the software you will produce and release. Having powerful tools like these can save a lot of time.

*testrepository* also integrates with *setuptools* and deploys a `testr` command for it. This provides easier integration with *setup.py*-based workflows – you can, for example, document your entire project around *setup.py*. The command `setup.py testr` accepts a few options, such as `--testr-args` – which adds more options to the `testr` run, or `--coverage`, which will be covered in the next section.

## 6.6 Coverage

Code coverage is a tool which complements unit testing. It uses code analysis tools and tracing hooks to determine which lines of your code have been executed; when used during a unit test run, it can show you which parts of your code base have been crossed over and which parts have not.

Writing tests is useful; but having a way to know what part of your code you may have missed is the cherry on the cake.

Obviously, the first thing to do is to install the **`coverage`** Python module on your system. Once this is done you will have access to the `coverage` program command from your shell.[2]

Using `coverage` in standalone mode is straightforward, and can be useful- it could lead you to part of your programs that are never run, and which might be "dead code". In addition, using it while your unit tests are running provides an obvious benefit: you'll know which parts of the code are not being tested. The test tools

[2]The command may also be named `python-coverage`, if you install `coverage` through your operating system installation software. That is the case on *Debian*, for example.

we've talked about so far are all integrated with `coverage`.

When using *nose*, you only need to add a few option switches to generate a nice code coverage output:

**Example 6.15** Using `nosetests --with-coverage`

```
$ nosetests --cover-package=ceilometer --with-coverage tests/test_pipeline ↩
   .py
.........................................
Name                                  Stmts   Miss  Cover   Missing
ceilometer                                0      0   100%
ceilometer.pipeline                     152     20    87%   49, 59, 113,  ↩
   127-128, 188-192, 275-280, 350-362
ceilometer.publisher                     12      3    75%   32-34
ceilometer.sample                        31      4    87%   81-84
ceilometer.transformer                   15      3    80%   26-32, 35
ceilometer.transformer.accumulator       17      0   100%
ceilometer.transformer.conversions       59      0   100%

TOTAL                                   888    393    56%


----------------------------------------------------------------
Ran 46 tests in 0.170s

OK
```

Adding the `--cover-package` option is important, since otherwise you will see **every** Python package used, including standard library or third-party modules. The output includes the lines of code that were not run – and which therefore have no tests. All you need to do now is spawn your favorite text editor and start writing some.

But you can do better, and make *coverage* generate nice HTML reports. Simply add the `--cover-html` flag, and the `cover` directory from which you ran the command will be populated with HTML pages. Each page will show you which parts of your source code were or were not run.

Coverage for **ceilometer.publisher** : 75%

12 statements | 9 run | 3 missing | 0 excluded

```
1  # -*- encoding: utf-8 -*-
2  #
3  # Copyright © 2013 Intel Corp.
4  # Copyright © 2013 eNovance
5  #
6  # Author: Yunhong Jiang <yunhong.jiang@intel.com>
7  #         Julien Danjou <julien@danjou.info>
8  #
9  # Licensed under the Apache License, Version 2.0 (the "License"); you may
10 # not use this file except in compliance with the License. You may obtain
11 # a copy of the License at
12 #
13 #      http://www.apache.org/licenses/LICENSE-2.0
14 #
15 # Unless required by applicable law or agreed to in writing, software
16 # distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
17 # WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
18 # License for the specific language governing permissions and limitations
19 # under the License.
20
21 import abc
22 from stevedore import driver
23 from ceilometer.openstack.common import network_utils
24
25
26 def get_publisher(url, namespace='ceilometer.publisher'):
27     """Get publisher driver and load it.
28
29     :param URL: URL for the publisher
30     :param namespace: Namespace to use to look for drivers.
31     """
32     parse_result = network_utils.urlsplit(url)
33     loaded_driver = driver.DriverManager(namespace, parse_result.scheme)
34     return loaded_driver.driver(parse_result)
35
36
37 class PublisherBase(object):
38     """Base class for plugins that publish the sampler."""
39
40     __metaclass__ = abc.ABCMeta
41
42     def __init__(self, parsed_url):
43         pass
44
45     @abc.abstractmethod
46     def publish_samples(self, context, samples):
47         "Publish samples into final conduit."
48
```

Figure 6.1: Coverage of `ceilometer.publisher`

If you want to be that guy, you can use the option `--cover-min-percentage=COVE`

R_MIN_PERCENTAGE, which will make the test suite fail if a minimum percentage of the code is not executed when the test suite is run.

**Warning**

A code coverage score of 100% doesn't necessarily mean that the code is entirely tested and that you can rest. It only proves that your whole code path has been run; there is no indication that every possible condition has been tested. So while being a respectable goal, it doesn't indicate anything conclusive.

When using `testrepository`, `coverage` can be run using *setuptools* integration.

**Example 6.16** Using coverage with *testrepository*

```
$ python setup.py testr --coverage
```

This will automatically run your test suite with *coverage* and generate an HTML report in the `cover` directory.

You should then use this information to consolidate your test suite and add tests for any code that is currently not being run. This is important; it facilitates later project maintenance, and increases your code's overall quality.

## 6.7 Using virtualenv with tox

In Chapter 5, the use of virtual environments is presented and discussed. One of their main uses is to provide a clean environment for running unit tests. It would be really sad if you thought that your tests were working, when in fact you were not, for example, respecting the dependency list.

You could write a script to deploy a virtual environment, install `setuptools`, and then install all of the dependencies required for both your application/library runtime and unit tests. But this is such a common use case that an application dedicated to this task has already been built: **tox**.

*Tox* aims to automate and standardize how tests are run in Python. To that end, it provides everything needed to run an entire test suite in a clean virtual environment, while also installing your application to check that the installation works fine.

Before using *tox*, you need to provide a configuration file. This file is named `tox.ini` and should be placed in the root directory of your project, beside your `setup.py` file.

```
$ touch tox.ini
```

You can now run tox successfully:

```
% tox
GLOB sdist-make: /home/jd/project/setup.py
python create: /home/jd/project/.tox/python
python inst: /home/jd/project/.tox/dist/project-1.zip
____________________ summary ____________________
  python: commands succeeded
  congratulations :)
```

Obviously this alone is not very useful. In this instance, `tox` creates a virtual environment in *.tox/python* using its default Python version, uses `setup.py` to create a distribution of your package and then installs it inside this virtual environment. No commands are then run, because we didn't specify any in the configuration file.

We can change this default behaviour by adding a command that will be run inside our test environment. Editing tox.ini to include the following:

```
[testenv]
commands=nosetests
```

will run the command `nosetests` will likely fail, since we don't have `nosetests` installed in the virtual environment. We need to list it as part of the dependencies to be installed.

```
[testenv]
deps=nose
commands=nosetests
```

When run, `tox` will now recreate the environment, install the new dependency and run the command `nosetests`, which will execute all of our unit tests. Obviously, we might want to add more dependencies – you can list them in the *deps* configuration option, but you can also use the `-rfile` syntax to read from a file. If you're using *pbr* to manage your *setup.py* file, you know that it reads the dependencies from a file called `requirements.txt`. It is therefore a good idea to tell *tox* to use that file too:

```
[testenv]
deps=nose
     -rrequirements.txt
commands=nosetests
```

The `[testenv]` section of the file defines the parameters for all virtual environments managed by `tox`. But as mentioned previously, *tox* can manage multiple Python virtual environments – indeed, it's possible to run our tests under a Python version other than the default one by passing the `-e` flag to `tox`:

```
% tox -e py26
GLOB sdist-make: /home/jd/project/setup.py
py26 create: /home/jd/project/.tox/py26
py26 installdeps: nose
py26 inst: /home/jd/project/.tox/dist/rebuildd-1.zip
py26 runtests: commands[0] | nosetests
.......

----------------------------------------------------------
```

```
Ran 7 tests in 0.029s


OK
_____________________ summary _____________________
  py26: commands succeeded
  congratulations :)
```

By default, tox can simulate many environments: *py24*, *py25*, *py26*, *py27*, *py30*, *py31*, *py32*, *py33*, *jython* and *pypy*! You can even add your own. To add an environment or to create a new one, you just need to add another section named `[testenv:_envname_]`. If we want to run a different command for one of the environments, it's easy with the following `tox.ini` file:

```
[testenv]
deps=nose
commands=nosetests

[testenv:py27]
commands=pytest
```

This only overrides the commands for the `py27` environment; so *nose* will still be installed as part of the dependencies when running `tox -e py27`, but the command `pytest` will be run instead.

We can create a new environment with an unsupported version of Python right away:

```
[testenv]
deps=nose
commands=nosetests

[testenv:py21]
basepython=python2.1
```

We can now (attempt to) use Python 2.1 to run our test suite – although I don't think it will work.

Now, it is likely that you will want to support multiple Python versions. So it would be great to have `tox` run all the tests for all the Python versions you want to support by default. This can be done by specifying the environment list you want to use when `tox` is run without arguments:

```
[tox]
envlist=py26,py27,py33,pypy

[testenv]
deps=nose
commands=nosetests
```

When `tox` is launched without any further arguments, all four environments listed will be created, populated with the dependencies and the application, and then the command `nosetests` will be run.

We can also use `tox` to integrate other tests like `flake8`, as discussed in Section 1.4.

```
[tox]
envlist=py26,py27,py33,pypy,pep8

[testenv]
deps=nose
commands=nosetests

[testenv:pep8]
deps=flake8
commands=flake8
```

In this case, the *pep8* environment will be run using the default version of Python,

which is probably fine.[3]

**Tip**

When running `tox`, you will spot that all of the environments are built and run in sequence. This can often make the process very long. Since the virtual environments are isolated, nothing prevents you from running `tox` commands in parallel. This is exactly what the detox package does, by providing a `detox` command which runs all of the default environments from *envlist* in parallel. You should *pip install* it!

## 6.8 Testing policy

Having testing code embedded in your project is wonderful, but how you run it is also extremely important. There are too many projects that have test code which lays around, but which fails to be run for some reason.

While this topic is not strictly limited to Python, I consider it important enough to emphasize here: you should have a zero tolerance policy on untested code. No code should be merged unless there is a proper set of unit tests to cover it.

The minimum that you should aim for is to be sure that each of the commits you push pass all the tests. Having an automated way to do that is even better.

For example, *OpenStack* relies on a specific workflow based on Gerrit, Jenkins and Zuul. Each commit pushed goes through the code review system provided by Gerrit, and Zuul is in charge of running a set of testing jobs against it using Jenkins. Jenkins runs the unit testing, and various higher-level functional tests for each project. This ensures that the submitted patches pass all tests. Code reviewing by a couple of developers makes sure that all code that is committed has associated unit tests.

If you are using the popular GitHub hosting service, Travis CI provides a way to run a test after each push or merge, or against pull requests that are submitted. While it is

[3] You can still specify the `basepython` option if you want to change that

unfortunate that this done post-push, it's still a fantastic way to track regressions. Travis supports all significant Python versions out of the box, and it's possible to customize it to a high degree. Once you've activated Travis on your project via their Web interface, adding a file is simple: `.travis.yml` does the job for you.

**Example 6.17** A `.travis.yml` example file

```
language: python
python:
  - "2.7"
  - "3.3"
# command to install dependencies
install: "pip install -r requirements.txt --use-mirrors"
# command to run tests
script: nosetests
```

Wherever your code is hosted, these days it is always possible to aim for some sort of automatic testing of your software, and to make sure that you are going forward with your project – not going backward by adding more bugs.

## 6.9 Interview with Robert Collins

You may have already used one of Robert's programs, without knowing – he is, among other things, the original author of the *Bazaar* distributed version control system. Today, he is a *Distinguished Technologist* at HP Cloud Services, where he works on OpenStack. Robert has written a lot of the Python tools described in this book, such as *fixtures*, *testscenarios*, *testrepository* and even *python-subunit*.

**What kind of testing policy would you advise using? When is it acceptable not to test code?**

I think it's an engineering trade-off – considering the likelihood of failure slipping through to production undetected, the cost of an undetected failure of that component, the size and cohesion of the team doing the work… Take OpenStack – 1600 contributors – a nuanced policy is very hard to work with there, as so many people have opinions. Generally speaking, there should be some automated check as part of landing in trunk that the code will do what it is intended to do **and** that what it is intended to do is what is needed. Often that speaks to requiring functional tests that might be in different code bases. Unit tests are great for speed and pinning down corner cases. I think it's ok to vary the balance between styles of testing, as long as there is testing.

Where the cost of testing is very high and the returns are very low, I think it's fine to make an informed decision not to test, but that's a relatively rare situation: most things can be tested fairly cheaply, and the benefit of catching errors early is usually quite high.

**What are the best strategies to put in place when writing Python code in order to make testing easier, and improve its quality?**

Separate out concerns – don't do multiple things in one place; this makes reuse easier, and that makes it easier to put test doubles in place. Take a pure functional approach when you can (e.g. in a single method either calculate something, or change some state, but where possible avoid doing

both). That way you can test all of the calculating behaviour without dealing with state changes – such as writing to a database, talking to an HTTP server, etc. The benefit works the other way around too – you can replace the calculation logic for tests to provoke corner case behaviour and detect via mocks / test doubles that the expected state propagation happens as desired. The most heinous stuff to test IME is deeply layered stacks with complex cross-layer behavioural dependencies. There you want to evolve the code so that the contract between layers is simple, predictable, and most usefully for testing – replaceable.

**In your opinion, what's the best way to organize unit tests in source code?**

Having a hierarchy like `$ROOT/$PACKAGE/tests` – but I do just one for a whole source tree (vs e.g. `$ROOT/$PACKAGE/$SUBPACKAGE/tests`).

Within tests, I often mirror the structure of the rest of the source tree: `$ROOT/$PACKAGE/foo.py` would be tested in `$ROOT/$PACKAGE/tests/test_foo.py`.

There should be no imports from tests by the rest of the tree except perhaps a test_suite/load_tests function in the top level `__init__`. This permits easily detaching the tests for small footprint installations.

**What are the tools that can be employed to build functional tests in Python?**

I just use whichever flavour of `unittest` is in use in the project: it's sufficiently flexible (particularly with things like `testresources` and parallel runners) to cater for most needs.

**How do you envision the future of unit testing libraries and frameworks in Python?**

The big challenges I see are:

- the continued expansion of parallel capabilities in new machines – 4 CPU phones now. Existing unit test internal APIs aren't optimised for parallel workloads. My *StreamResult* work is aimed directly at this;
- more complex scheduling support – a less ugly solution for the problems that class and module scoped setup aim at;
- finding some way to consolidate the large variety of frameworks we have today: it would be great to be able to get a consolidated view across multiple projects – for integration testing – that have different test runners in use.