6

UNIT TESTING

Many find unit testing to be arduous and time-consuming, and some people and projects have no testing policy. This chapter assumes that you see the wisdom of unit testing! Writing code that is not tested is fundamentally useless, as there's no way to conclusively prove that it works. If you need convincing, I suggest you start by reading about the benefits of test-driven development.

In this chapter you'll learn about the Python tools you can use to construct a comprehensive suite of tests that will make testing simpler and more automated. We'll talk about how you can use tools to make your software rock solid and regression-free. We'll cover creating reusable test objects, running tests in parallel, revealing untested code, and using virtual environments to make sure your tests are clean, as well as some other good-practice methods and ideas.

The Basics of Testing

Writing and running unit tests is uncomplicated in Python. The process is not intrusive or disruptive, and unit testing will greatly help you and other developers in maintaining your software. Here I'll discuss some of the absolute basics of testing that will make things easier for you.

Some Simple Tests

First, you should store tests inside a `tests` submodule of the application or library they apply to. Doing so will allow you to ship the tests as part of your module so that they can be run or reused by anyone—even after your software is installed—without necessarily using the source package. Making the tests a submodule of your main module also prevents them from being installed by mistake in a top-level `tests` module.

Using a hierarchy in your test tree that mimics the hierarchy of your module tree will make the tests more manageable. This means that the tests covering the code of *mylib/foobar.py* should be stored inside *mylib/tests/test_foobar.py*. Consistent nomenclature makes things simpler when you're looking for the tests related to a particular file. Listing 6-1 shows the simplest unit test you can write.

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

Listing 6-1: A really simple test in test_true.py

This will simply assert that the behavior of the program is what you expect. To run this test, you need to load the *test_true.py* file and run the `test_true()` function defined within.

However, writing and running an individual test for each of your test files and functions would be a pain. For small projects with simple usage, the `pytest` package comes to the rescue—once installed via `pip`, pytest provides the `pytest` command, which loads every file whose name starts with *test_* and then executes all functions within that start with `test_`.

With just the *test_true.py* file in our source tree, running `pytest` gives us the following output:

```
 $ pytest -v test_true.py
========================== test session starts ==========================
platform darwin -- Python 3.6.4, pytest-3.3.2, py-1.5.2, pluggy-0.6.0 --
/usr/local/opt/python/bin/python3.6
cachedir: .cache
rootdir: examples, inifile:
collected 1 item

test_true.py::test_true PASSED                                      [100%]

======================== 1 passed in 0.01 seconds ========================
```

The -v option tells pytest to be verbose and print the name of each test run on a separate line. If a test fails, the output changes to indicate the failure, accompanied by the whole traceback.

Let's add a failing test this time, as shown in Listing 6-2.

```
def test_false():
    assert False
```

Listing 6-2: A failing test in test_true.py

If we run the test file again, here's what happens:

```
 $ pytest -v test_true.py
========================== test session starts ==========================
platform darwin -- Python 3.6.4, pytest-3.3.2, py-1.5.2, pluggy-0.6.0 -- /usr/
local/opt/python/bin/python3.6
cachedir: .cache
rootdir: examples, inifile:
collected 2 items

test_true.py::test_true PASSED                                      [ 50%]
test_true.py::test_false FAILED                                     [100%]

================================ FAILURES ================================
_________________________________ test_false _________________________________

    def test_false():
>       assert False
E       assert False

test_true.py:5: AssertionError
=================== 1 failed, 1 passed in 0.07 seconds ===================
```

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

Simple, isn't it? While simplistic, a lot of small projects use this approach and it works very well. Those projects require no tools or libraries other than pytest and thus can rely on simple assert tests.

As you start to write more sophisticated tests, pytest will help you understand what's wrong in your failing tests. Imagine the following test:

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

When pytest is run, it gives the following output:

```
 $ pytest test_true.py
========================== test session starts ==========================
```

```
platform darwin -- Python 3.6.4, pytest-3.3.2, py-1.5.2, pluggy-0.6.0
rootdir: /Users/jd/Source/python-book/examples, inifile:
plugins: celery-4.1.0
collected 1 item

test_true.py F                                                       [100%]

================================ FAILURES ================================
_________________________________ test_key _________________________________

    def test_key():
        a = ['a', 'b']
        b = ['b']
>       assert a == b
E       AssertionError: assert ['a', 'b'] == ['b']
E         At index 0 diff: 'a' != 'b'
E         Left contains more items, first extra item: 'b'
E         Use -v to get the full diff

test_true.py:10: AssertionError
======================== 1 failed in 0.07 seconds ========================
```

This tells us that `a` and `b` are different and that this test does not pass. It also tells us exactly how they are different, making it easy to fix the test or code.

Skipping Tests

If a test cannot be run, you will probably want to skip that test—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 use the `pytest.skip()` function, which will mark the test as skipped and move on to the next one. The `pytest.mark.skip` decorator skips the decorated test function unconditionally, so you'll use it when a test always needs to be skipped. Listing 6-3 shows how to skip a test using these methods.

```
import pytest

try:
    import mylib
except ImportError:
    mylib = None

@pytest.mark.skip("Do not run this")
def test_fail():
    assert False

@pytest.mark.skipif(mylib is None, reason="mylib is not available")
def test_mylib():
    assert mylib.foobar() == 42
```

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

Listing 6-3: Skipping tests

When executed, this test file will output the following:

```
 $ pytest -v examples/test_skip.py
========================== test session starts ==========================
platform darwin -- Python 3.6.4, pytest-3.3.2, py-1.5.2, pluggy-0.6.0 -- /usr/
local/opt/python/bin/python3.6
cachedir: .cache
rootdir: examples, inifile:
collected 3 items

examples/test_skip.py::test_fail SKIPPED
[ 33%]
examples/test_skip.py::test_mylib SKIPPED
[ 66%]
examples/test_skip.py::test_skip_at_runtime SKIPPED
[100%]

================= 3 skipped in 0.01 seconds =================
```

The output of the test run in Listing 6-3 indicates that, in this case, all the tests have been skipped. This information allows you to ensure you didn't accidentally skip a test you expected to run.

Running Particular Tests

When using `pytest`, you often want to run only a particular subset of your tests. You can select which tests you want to run by passing their directory or files as an argument to the `pytest` command line. For example, calling `pytest test_one.py` will only run the *test_one.py* test. Pytest also accepts a directory as argument, and in that case, it will recursively scan the directory and run any file that matches the *test_*.py* pattern.

You can also add a filter with the `-k` argument on the command line in order to execute only the test matching a name, as shown in Listing 6-4.

```
$ pytest -v examples/test_skip.py -k test_fail
========================== test session starts ==========================
platform darwin -- Python 3.6.4, pytest-3.3.2, py-1.5.2, pluggy-0.6.0 -- /usr/
local/opt/python/bin/python3.6
cachedir: .cache
rootdir: examples, inifile:
collected 3 items

examples/test_skip.py::test_fail SKIPPED
[100%]
```

```
=== 2 tests deselected ===
=== 1 skipped, 2 deselected in 0.04 seconds ===
```

Listing 6-4: Filtering tests run by name

Names are not always the best way to filter which tests will run. Commonly, a developer would group tests by functionalities or types instead. Pytest provides a dynamic marking system that allows you to mark tests with a keyword that can be used as a filter. To mark tests in this way, use the `-m` option. If we set up a couple of tests like this:

```
import pytest

@pytest.mark.dicttest
def test_something():
    a = ['a', 'b']
    assert a == a

def test_something_else():
    assert False
```

we can use the `-m` argument with `pytest` to run only one of those tests:

```
$ pytest -v test_mark.py -m dicttest
=== test session starts ===
platform darwin -- Python 3.6.4, pytest-3.3.2, py-1.5.2, pluggy-0.6.0 -- /usr/
local/opt/python/bin/python3.6
cachedir: .cache
rootdir: examples, inifile:
collected 2 items

test_mark.py::test_something PASSED
[100%]

=== 1 tests deselected ===
=== 1 passed, 1 deselected in 0.01 seconds ===
```

The `-m` marker accepts more complex queries, so we can also run all tests that are *not* marked:

```
$ pytest test_mark.py -m 'not dicttest'
=== test session starts ===
platform darwin -- Python 3.6.4, pytest-3.3.2, py-1.5.2, pluggy-0.6.0
rootdir: examples, inifile:
collected 2 items

test_mark.py F
[100%]

=== FAILURES ===
test_something_else
```

```
    def test_something_else():
>       assert False
E       assert False

test_mark.py:10: AssertionError
=== 1 tests deselected ===
=== 1 failed, 1 deselected in 0.07 seconds ===
```

Here pytest executed every test that was not marked as `dicttest`—in this case, the `test_something_else` test, which failed. The remaining marked test, `test_something`, was not executed and so is listed as `deselected`.

Pytest accepts complex expressions composed of the `or`, `and`, and `not` keywords, allowing you to do more advanced filtering.

Running Tests in Parallel

Test suites can take a long time to run. It's not uncommon for a full suite of unit tests to take tens of minutes to run in large software projects. By default, pytest runs all tests serially, in an undefined order. Since most computers have several CPUs, you can usually speed things up if you split the list of tests and run them on multiple CPUs.

To handle this approach, pytest provides the plugin `pytest-xdist`, which you can install with `pip`. This plugin extends the pytest command line with the `--numprocesses` argument (shortened as `-n`), which accepts as its argument the number of CPUs to use. Running `pytest -n 4` would run your test suite using four parallel processes, balancing the load across the available CPUs.

Because the number of CPUs can change from one computer to another, the plugin also accepts the `auto` keyword as a value. In this case, it will probe the machine to retrieve the number of CPUs available and start this number of processes.

Creating Objects Used in Tests with Fixtures

In unit testing, you'll often need to execute a set of common instructions before and after running a test, and those instructions will use certain components. For example, you might need an object that represents the configuration state of your application, and you'll likely want that object to be initialized before each test, then reset to its default values when the test is achieved. Similarly, if your test relies on the temporary creation of a file, the file must be created before the test starts and deleted once the test is done. These components, known as *fixtures*, are set up before a test and cleaned up after the test has finished.

With pytest, fixtures are defined as simple functions. The fixture function should return the desired object(s) so that a test using that fixture can use that object.

Here's a simple fixture:

```
import pytest

@pytest.fixture
def database():
    return <some database connection>

def test_insert(database):
    database.insert(123)
```

The database fixture is automatically used by any test that has `database` in its argument list. The `test_insert()` function will receive the result of the `database()` function as its first argument and use that result as it wants. When we use a fixture this way, we don't need to repeat the database initialization code several times.

Another common feature of code testing is tearing down after a test has used a fixture. For example, you may need to close a database connection. Implementing the fixture as a generator allows us to add teardown functionality, as shown in Listing 6-5.

```
import pytest

@pytest.fixture
def database():
    db = <some database connection>
    yield db
    db.close()

def test_insert(database):
    database.insert(123)
```

Listing 6-5: Teardown functionality

Because we used the `yield` keyword and made `database` a generator, the code after the `yield` statement runs when the test is done. That code will close the database connection at the end of the test.

However, closing a database connection for each test might impose an unnecessary runtime cost, as tests may be able to reuse that same connection. In that case, you can pass the `scope` argument to the fixture decorator, specifying the scope of the fixture:

```
import pytest

@pytest.fixture(scope="module")
def database():
    db = <some database connection>
    yield db
    db.close()

def test_insert(database):
    database.insert(123)
```

By specifying the `scope="module"` parameter, you initialize the fixture once for the whole module, and the same database connection will be passed to all test functions requesting a database connection.

Finally, you can run some common code before and after your tests by marking fixtures as *automatically used* with the `autouse` keyword, rather than specifying them as an argument for each of the test functions. Specifying the `autouse=True` keyword argument to the `pytest.fixture()` function will make sure the fixture is called before running any test in the module or class it is defined in, as in this example:

```
import os

import pytest

@pytest.fixture(autouse=True)
def change_user_env():
    curuser = os.environ.get("USER")
    os.environ["USER"] = "foobar"
    yield
    os.environ["USER"] = curuser

def test_user():
    assert os.getenv("USER") == "foobar"
```

Such automatically enabled features are handy, but make sure not to abuse fixtures: they are run before each and every test covered by their scope, so they can slow down a test run significantly.

Running Test Scenarios

When unit testing, you may want to run the same error-handling test with several different objects that trigger that error, or you may want to run an entire test suite against different drivers.

We relied heavily on this latter approach when developing *Gnocchi*, a time series database. Gnocchi provides an abstract class that we call the *storage API*. Any Python class can implement this abstract base 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, we need a class of unit tests that runs against each driver—thus running against each implementation of this storage API—to be sure all drivers conform to what the callers expect.

An easy way to achieve this is by using *parameterized fixtures*, which will run all the tests that use them several times, once for each of the defined parameters. Listing 6-6 shows an example of using parameterized fixtures to run a single test twice with different parameters: once for `mysql` and once for `postgresql`.

```
import pytest
import myapp
```

```
@pytest.fixture(params=["mysql", "postgresql"])
def database(request):
    d = myapp.driver(request.param)
    d.start()
    yield d
    d.stop()

def test_insert(database):
    database.insert("somedata")
```

Listing 6-6: Running a test using parameterized fixtures

In Listing 6-6, the `driver` fixture is parameterized with two different values, each the name of a database driver that is supported by the application. When `test_insert` is run, it is actually run twice: once with a MySQL database connection and once with a PostgreSQL database connection. This allows us to easily reuse the same test with different scenarios, without adding many lines of code.

Controlled Tests Using Mocking

Mock objects are simulated objects that mimic the behavior of real application objects, but in particular and controlled ways. These are especially useful in creating environments that describe precisely the conditions for which you would like to test code. You can replace all objects but one with mock objects to isolate the behavior of your focus object and create an enviroment for testing your code.

One use case is in writing an HTTP client, since it is likely impossible (or at least extremely complicated) to spawn the HTTP server and test it through all scenarios to return every possible value. HTTP clients are especially difficult to test for all failure scenarios.

The standard library for creating mock objects in Python is `mock`. Starting with Python 3.3, `mock` has been merged into the Python Standard Library as `unittest.mock`. You can, therefore, use a snippet like the following to maintain backward compatibility between Python 3.3 and earlier versions:

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

The `mock` library is pretty simple to use. Any attribute accessed on a `mock.Mock` object is dynamically created at runtime. Any value can be set to such an attribute. Listing 6-7 shows `mock` being used to create a fake object with a fake attribute.

```
>>> from unittest import mock
>>> m = mock.Mock()
>>> m.some_attribute = "hello world"
```

```
>>> m.some_attribute
"hello world"
```

Listing 6-7: Accessing the `mock.Mock` attribute

You can also dynamically create a method on a malleable object, as in Listing 6-8 where we create a fake method that always returns 42 and accepts anything as an argument.

```
>>> from unittest import mock
>>> m = mock.Mock()
>>> m.some_method.return_value = 42
>>> m.some_method()
42
>>> m.some_method("with", "arguments")
42
```

Listing 6-8: Creating methods on a `mock.Mock` object

In just a few lines, your `mock.Mock` object now has a `some_method()` method that returns 42. It accepts any kind of argument, and there is no check on what the values are—yet.

Dynamically created methods can also have (intentional) side effects. Rather than being boilerplate methods that just return a value, they can be defined to execute useful code.

Listing 6-9 creates a fake method that has the side effect of printing the `"hello world"` string.

```
>>> from unittest import mock
>>> m = mock.Mock()
>>> def print_hello():
...     print("hello world!")
...     return 43
...
❶ >>> m.some_method.side_effect = print_hello
>>> m.some_method()
hello world!
43
❷ >>> m.some_method.call_count
1
```

Listing 6-9: Creating methods on a `mock.Mock` object with side effects

We assign an entire function to the `some_method` attribute ❶. This technique allows us to implement more complex scenarios in a test because we can plug any code needed for testing into a mock object. We then just need to pass this mock object to whichever function expects it.

The `call_count` attribute ❷ is a simple way of checking the number of times a method has been called.

The mock library uses the action/assertion pattern: this means that once your test has run, it's up to you to check that the actions you are mocking were correctly executed. Listing 6-10 applies the assert() method to our mock objects to perform these checks.

```
>>> from unittest import mock
>>> m = mock.Mock()
❶ >>> m.some_method('foo', 'bar')
<Mock name='mock.some_method()' id='26144272'>
❷ >>> m.some_method.assert_called_once_with('foo', 'bar')
>>> m.some_method.assert_called_once_with('foo', ❸mock.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')
```

Listing 6-10: Checking method calls

We create a method with the arguments foo and bar to stand in as our tests by calling the method ❶. The usual way to check calls to a mock object is to use the assert_called() methods, such as assert_called_once_with() ❷. To these methods, you need to pass the values that you expect callers to use when calling your mock method. If the values passed are not the ones being used, then mock raises an AssertionError. If you don't know what arguments may be passed, you can use mock.ANY as a value ❸; that will match any argument passed to your mock method.

Th mock library can also be used to patch some function, method, or object from an external module. In Listing 6-11, we replace the os.unlink() function with a fake function we provide.

```
>>> from unittest 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!
```

Listing 6-11: Using mock.patch

When used as a context manager, `mock.patch()` replaces the target function with the function we provide so the code executed inside the context uses that patched method. With the `mock.patch()` method, it's possible to change any part of an external piece of code, making it behave in a way that lets you test all conditions in your application, as shown in Listing 6-12.

```
from unittest import mock

import pytest
import requests

class WhereIsPythonError(Exception):
    pass

❶ 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)

❷ @mock.patch('requests.get', get_fake_get(
    200, 'Python is a programming language for sure'))
def test_python_is():
    assert is_python_still_a_programming_language() is True

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

@mock.patch('requests.get', get_fake_get(404, 'Whatever'))
def test_bad_status_code():
    with pytest.raises(WhereIsPythonError):
        is_python_still_a_programming_language()

@mock.patch('requests.get', raise_get)
def test_ioerror():
```

```
    with pytest.raises(WhereIsPythonError):
        is_python_still_a_programming_language()
```

Listing 6-12: Using `mock.patch()` to test a set of behaviors

Listing 6-12 implements a test suite that searches for all instances of the string "Python is a programming language" on the *http://python.org/* web page ❶. There is no way to test negative scenarios (where this sentence is not on the web page) without modifying the page itself—something we're not able to do, obviously. In this case, we're using `mock` to cheat and change the behavior of the request so it returns a mocked reply with a fake page that doesn't contain that string. This allows us to test the negative scenario in which *http://python.org/* does not contain this sentence, making sure the program handles that case correctly.

This example uses the decorator version of `mock.patch()` ❷. Using the decorator does not change the mocking behavior, but it is simpler when you need to use mocking within the context of an entire test function.

Using mocking, we can simulate any problem, such as a web server returning a 404 error, an I/O error, or a network latency issue. We can make sure code returns the correct values or raises the correct exception in every case, ensuring our code always behaves as expected.

Revealing Untested Code with coverage

A great complement to unit testing, the `coverage` tool identifies whether any of your code has been missed during 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 codebase 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 during the testing process is the cherry on the cake.

Install the `coverage` Python module on your system via `pip` to have access to the `coverage` program command from your shell.

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

Using `coverage` in stand-alone mode is straightforward. It can show you parts of your programs that are never run and which code might be "dead code," that is, code that could be removed without modifying the normal workflow of the program. All the test tools we've talked about so far in this chapter are integrated with `coverage`.

When using `pytest`, just install the `pytest-cov` plugin via `pip install pytest-pycov` and add a few option switches to generate a detailed code coverage output, as shown in Listing 6-13.

```
$ pytest --cov=gnocchiclient gnocchiclient/tests/unit
---------- coverage: platform darwin, python 3.6.4-final-0 -----------
Name                                      Stmts   Miss Branch BrPart  Cover
---------------------------
gnocchiclient/__init__.py                     0      0      0      0   100%
gnocchiclient/auth.py                        51     23      6      0    49%
gnocchiclient/benchmark.py                  175    175     36      0     0%
--snip--
---------------------------
TOTAL                                      2040   1868    424      6     8%

=== passed in 5.00 seconds ===
```

Listing 6-13: Using coverage with pytest

The `--cov` option enables the coverage report at the end of the test run. You need to pass the package name as an argument for the plugin to filter the coverage report properly. The output includes the lines of code that were not run and therefore have no tests. All you need to do now is spawn your favorite text editor and start writing tests for that code.

However, `coverage` goes one better, allowing you to generate clear HTML reports. Simply add the `--cov-report=html` flag, and the *htmlcov* 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.

If you want to be *that* person, you can use the option `--cover-fail-under=COVER_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. While having a good coverage percentage is a decent goal, and while the tool is useful to gain insight into the state of your test coverage, defining an arbitrary percentage value does not provide much insight. Figure 6-1 shows an example of a coverage report with the percentage at the top.

For example, a code coverage score of 100 percent is a respectable goal, but it does not necessarily mean the code is entirely tested and 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.

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

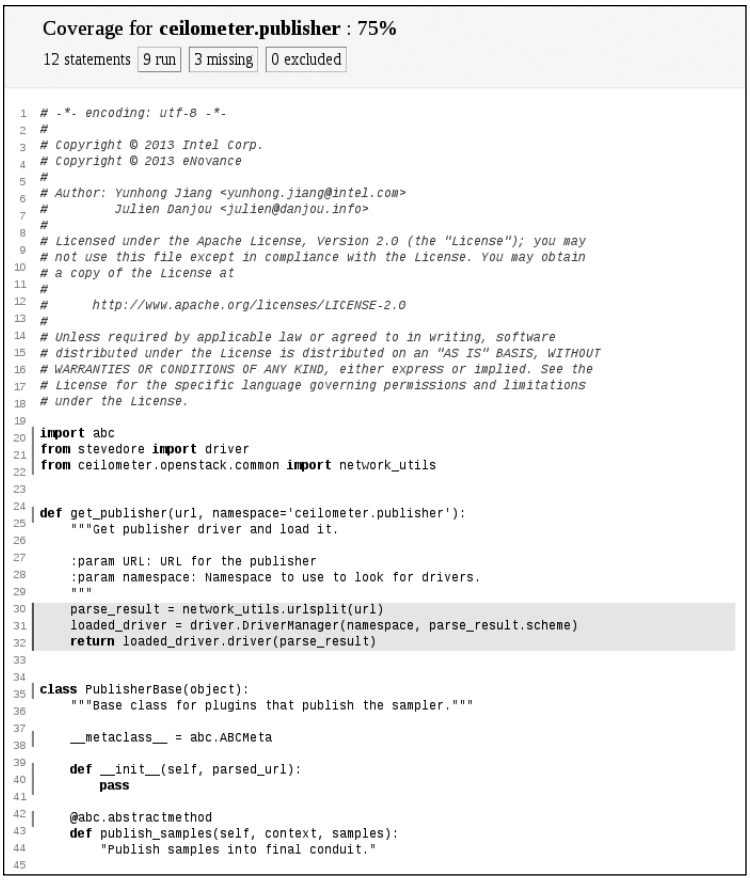

Figure 6-1: Coverage of `ceilometer.publisher`

Virtual Environments

Earlier we mentioned the danger that your tests may not capture the absence of dependencies. Any application of significant size inevitably depends on external libraries to provide features the application needs, but there are many ways external libraries might cause issues on your operating system. Here are a few:

- Your system does not have the library you need packaged.
- Your system does not have the right *version* of the library you need packaged.
- You need two different versions of the same library for two different applications.

These problems can happen when you first deploy your application or later on, while it's running. Upgrading a Python library installed via your

system manager might break your application in a snap without warning, for reasons as simple as an API change in the library being used by the application.

The solution is for each application to use a library directory that contains all the application's dependencies. This directory is then used to load the needed Python modules rather than the system-installed ones.

Such a directory is known as a *virtual environment*.

Setting Up a Virtual Environment

The tool `virtualenv` handles virtual environments automatically for you. Until Python 3.2, you'll find it in the `virtualenv` package that you can install using `pip install virtualenv`. If you use Python 3.3 or later, it's available directly via Python under the `venv` name.

To use the module, load it as the main program with a destination directory as its argument, like so:

```
$ python3 -m venv myvenv
$ ls foobar
bin        include    lib        pyvenv.cfg
```

Once run, `venv` creates a *lib/pythonX.Y* directory and uses it to install `pip` into the virtual environment, which will be useful to install further Python packages.

You can then activate the virtual environment by "sourcing" the `activate` command. Use the following on Posix systems:

```
$ source myvenv/bin/activate
```

On Windows systems, use this code:

```
> \myvenv\Scripts\activate
```

Once you do that, your shell prompt should appear prefixed by the name of your virtual environment. Executing `python` will call the version of Python that has been copied into the virtual environment. You can check that it's working by reading the `sys.path` variable and checking that it has your virtual environment directory as its first component.

You can stop and leave the virtual environment at any time by calling the `deactivate` command:

```
$ deactivate
```

That's it. Also note that you are not forced to run `activate` if you want to use the Python installed in your virtual environment just once. Calling the `python` binary will also work:

```
$ myvenv/bin/python
```

Now, while we're in our activated virtual environment, we do not have access to any of the modules installed and available on the main system. That is the point of using a virtual environment, but it does mean we probably need to install the packages we need. To do that, use the standard `pip` command to install each package, and the packages will install in the right place, without changing anything about your system:

```
$ source myvenv/bin/activate
(myvenv) $ pip install six
Downloading/unpacking six
  Downloading six-1.4.1.tar.gz
  Running setup.py egg_info for package six

Installing collected packages: six
  Running setup.py install for six

Successfully installed six
Cleaning up...
```

Voilà! We can install all the libraries we need and then run our application from this virtual environment, without breaking our system. It's easy to see how we can script this to automate the installation of a virtual environment based on a list of dependencies, as in Listing 6-14.

```
virtualenv myappvenv
source myappvenv/bin/activate
pip install -r requirements.txt
deactivate
```

Listing 6-14: Automatic virtual environment creation

It can still be useful to have access to your system-installed packages, so `virtualenv` allows you to enable them when creating your virtual environment by passing the `--system-site-packages` flag to the `virtualenv` command.

Inside `myvenv`, you will find a *pyvenv.cfg*, the configuration file for this environment. It doesn't have a lot of configuration options by default. You should recognize `include-system-site-package`, whose purpose is the same as the `--system-site-packages` of `virtualenv` that we described earlier.

As you might guess, virtual environments are incredibly useful for automated runs of unit test suites. Their use is so widespread that a particular tool has been built to address it.

Using virtualenv with tox

One of the central uses of virtual environments is to provide a clean environment for running unit tests. It would be detrimental if you were under the impression that your tests were working, when they were not, for example, respecting the dependency list.

One way to ensure you're accounting for all the dependencies would be to 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. Luckily, this is such a popular use case that an application dedicated to this task has already been built: `tox`.

The `tox` management tool 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.

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

```
$ touch tox.ini
```

You can then 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 :)
```

In this instance, `tox` creates a virtual environment in *.tox/python* using the default Python version. It uses *setup.py* to create a distribution of your package, which it then installs inside this virtual environment. No commands are run, because we did not specify any in the configuration file. This alone is not particularly useful.

We can change this default behavior by adding a command to run inside our test environment. Edit *tox.ini* to include the following:

```
[testenv]
commands=pytest
```

Now `tox` runs the command `pytest`. However, since we do not have `pytest` installed in the virtual environment, this command will likely fail. We need to list `pytest` as a dependency to be installed:

```
[testenv]
deps=pytest
commands=pytest
```

When run now, `tox` re-creates the environment, installs the new dependency, and runs the command `pytest`, which executes all of the unit tests. To add more dependencies, you can either list them in the `deps` configuration option, as is done here, or use the `-rfile` syntax to read from a file.

Re-creating an Environment

Sometimes you'll need to re-create an environment to, for example, ensure things work as expected when a new developer clones the source code repository and runs `tox` for the first time. For this, `tox` accepts a `--recreate` option that will rebuild the virtual environment from scratch based on parameters you lay out.

You define the parameters for all virtual environments managed by `tox` in the `[testenv]` section of *tox.ini*. And, as mentioned, `tox` can manage multiple Python virtual environments—indeed, it is possible to run our tests under a Python version other than the default one by passing the `-e` flag to `tox`, like so:

```
 % 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] | pytests
 --snip--
== test session starts ==
=== 5 passed in 4.87 seconds ====
```

By default, `tox` simulates any environment that matches an existing Python version: `py24`, `py25`, `py26`, `py27`, `py30`, `py31`, `py32`, `py33`, `py34`, `py35`, `py36`, `py37`, `jython`, and `pypy`! Furthermore, you can define your own environments. You just need to add another section named `[testenv:_envname_]`. If you want to run a particular command for just one of the environments, you can do so easily by listing the following in the *tox.ini* file:

```
[testenv]
deps=pytest
commands=pytest

[testenv:py36-coverage]
deps={[testenv]deps}
      pytest-cov
commands=pytest --cov=myproject
```

By using `pytest --cov=myproject` under the `py36-coverage` section as shown here, you override the commands for the `py36-coverage` environment, meaning when you run `tox -e py36-coverage`, `pytest` is installed as part of the dependencies, but the command `pytest` is actually run instead with the coverage option. For that to work, the `pytest-cov` extension must be installed: to this end, we replace the `deps` value with the `deps` from `testenv` and add the `pytest-cov` dependency. Variable interpolation is also supported by `tox`, so you can refer to any other field from the *tox.ini* file and use it as a variable, the syntax being `{[env_name]variable_name}`. This allows us to avoid repeating the same things over and over again.

Using Different Python Versions

We can also create a new environment with an unsupported version of Python right away with the following in *tox.ini*:

```
[testenv]
deps=pytest
commands=pytest

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

When we run this, it will now (attempt to) use Python 2.1 to run the test suite—although since it is very unlikely you have this ancient Python version installed on your system, I doubt this would work for you!

It's likely that you'll want to support multiple Python versions, in which case it would be useful to have `tox` run all the tests for all the Python versions you want to support by default. You can do this by specifying the environment list you want to use when `tox` is run without arguments:

```
[tox]
envlist=py35,py36,pypy

[testenv]
deps=pytest
commands=pytest
```

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

Integrating Other Tests

We can also use `tox` to integrate tests like `flake8`, as discussed in Chapter 1. The following *tox.ini* file provides a PEP 8 environment that will install `flake8` and run it:

```
[tox]
envlist=py35,py36,pypy,pep8

[testenv]
deps=pytest
commands=pytest

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

In this case, the `pep8` environment is run using the default version of Python, which is probably fine, though you can still specify the `basepython` option if you want to change that.

When running `tox`, you'll notice that all the environments are built and run sequentially. This can make the process very long, but since 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 that runs all of the default environments from *envlist* in parallel. You should `pip install` it!

Testing Policy

Embedding testing code in your project is an excellent idea, but how that code is run is also extremely important. Too many projects have test code lying around that fails to run for some reason or other. This topic is not strictly limited to Python, but I consider it important enough to emphasize here: you should have a zero-tolerance policy regarding untested code. No code should be merged without a proper set of unit tests to cover it.

The minimum you should aim for is that each of the commits you push passes all the tests. Automating this process is even better. For example, OpenStack relies on a specific workflow based on *Gerrit* (a web-based code review service) and *Zuul* (a continuous integration and delivery service). Each commit pushed goes through the code review system provided by Gerrit, and Zuul is in charge of running a set of testing jobs. Zuul runs the unit tests and various higher-level functional tests for each project. This code review, which is executed by a couple of developers, makes sure all code committed has associated unit tests.

If you're using the popular GitHub hosting service, *Travis CI* is a tool that allows you to run tests after each push or merge or against pull requests that are submitted. While it is unfortunate that this testing is done post-push, it's still a fantastic way to track regressions. Travis supports all significant Python versions out of the box, and it can be customized significantly. Once you've activated Travis on your project via the web interface at *https://www.travis-ci.org/*, just add a *.travis.yml* file that will determine how the tests are run. Listing 6-15 shows an example of a *.travis.yml* file.

```
language: python
python:
  - "2.7"
  - "3.6"
# command to install dependencies
install: "pip install -r requirements.txt --use-mirrors"
# command to run tests
script: pytest
```

Listing 6-15: A .travis.yml *example file*

With this file in place in your code repository and Travis enabled, the latter will spawn a set of jobs to test your code with the associated unit tests. It's easy to see how you can customize this by simply adding dependencies and tests. Travis is a paid service, but the good news is that for open source projects, it's entirely free!

The tox-travis package (*https://pypi.python.org/pypi/tox-travis/*) is also worth looking into, as it will polish the integration between tox and Travis by running the correct tox target depending on the Travis environment being used. Listing 6-16 shows an example of a *.travis.yml* file that will install tox-travis before running tox.

```
sudo: false
language: python
python:
  - "2.7"
  - "3.4"
install: pip install tox-travis
script: tox
```

Listing 6-16: A .travis.yml *example file with* tox-travis

Using tox-travis, you can simply call tox as the script on Travis, and it will call tox with the environment you specify here in the *.travis.yml* file, building the necessary virtual environment, installing the dependency, and running the commands you specified in *tox.ini*. This makes it easy to use the same workflow both on your local development machine and on the Travis continuous integration platform.

These days, wherever your code is hosted, it is always possible to apply some automatic testing of your software and to make sure your project is moving forward, not being held back by the addition of bugs.

Robert Collins on Testing

Robert Collins 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 is also the author of many of the Python tools described in this book, such as fixtures, testscenarios, testrepository, and even python-subunit—you may have used one of his programs without knowing it!

What kind of testing policy would you advise using? Is it ever acceptable not to test code?

I think testing is an engineering trade-off: you must consider the likelihood of a failure slipping through to production undetected, the cost and size of an undetected failure, and cohesion of the team doing the work. Take OpenStack, which has 1,600 contributors: it's difficult to work with a nuanced policy with so many people with their own opinions. Generally speaking, a project needs some automated testing to check 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 requires functional tests that might be in different codebases. Unit tests are excellent for speed and pinning down corner cases. I think it is okay 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 situation is relatively rare: most things can be tested reasonably cheaply, and the benefit of catching errors early is usually quite high.

What are the best strategies when writing Python code to make testing manageable and improve the quality of the code?

Separate out concerns and don't do multiple things in one place; this makes reuse natural, and that makes it easier to put test doubles in place. Take a purely functional approach when possible; for example, in a single method either calculate something or change some state, but avoid doing both. That way you can test all of the calculating behaviors without dealing with state changes, such as writing to a database or talking to an HTTP server. The benefit works the other way around too—you can replace the calculation logic for tests to provoke corner case behavior and use mocks and test doubles to check that the expected state propagation happens as desired. The most heinous things to test are deeply layered stacks with complex cross-layer behavioral dependencies. There you want to evolve the code so that the contract between layers is simple, predictable, and—most usefully for testing—replaceable.

What's the best way to organize unit tests in source code?

Have a clear hierarchy, like *$ROOT/$PACKAGE/tests*. I tend to do just one hierarchy for a whole source tree, for example *$ROOT/$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*.

The rest of the tree should not import from the tests tree, except perhaps in the case of a `test_suite/load_tests` function in the top level `__init__`. This permits you to easily detach the tests for small-footprint installations.

What do you see as the future of unit-testing libraries and frameworks in Python?

The significant challenges I see are these:

- The continued expansion of parallel capabilities in new machines, like phones with four CPUs. Existing unit test internal APIs are not optimized for parallel workloads. My work on the StreamResult Java class is aimed directly at resolving 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 vast variety of frameworks we have today: for integration testing, it would be great to be able to get a consolidated view across multiple projects that have different test runners in use.