Chapter 44

Writing Basic Unit Tests

Difficulty

Newcomer

Skills

- All you need to know is some Python.

Problem/Task

As you know by now, Zope 3 gains its incredible stability from testing any code in great detail. The currently most common method is to write unit tests. This chapter introduces unit tests – which are Zope 3 independent – and introduces some of the subtleties.

Solution

44.1 Implementing the Sample Class

Before we can write tests, we have to write some code that we can test. Here, we will implement a simple class called `Sample` with a public attribute `title` and `description` that is accessed via `getDescription()` and mutated using `setDescription()`. Further, the description must be either a regular or unicode string.

Since this code will not depend on Zope, open a file named `test_sample.py` anywhere and add the following class:

```
Sample(object):
    """A trivial Sample object."""

    title = None

    def __init__(self):
        """Initialize object."""
        self._description = ''

```

```
10      def setDescription(self, value):
11          """Change the value of the description."""
12          assert isinstance(value, (str, unicode))
13          self._description = value
14
15      def getDescription(self):
16          """Change the value of the description."""
17          return self._description
```

▷ Line 4: The `title` is just publicly declared and a value of `None` is given. Therefore this is just a regular attribute.

▷ Line 8: The actual description string will be stored in `_description`.

▷ Line 12: Make sure that the description is only a regular or unicode string, like it was stated in the requirements.

If you wish you can now manually test the class with the interactive Python shell. Just start Python by entering `python` in your shell prompt. Note that you should be in the directory in which `test_sample.py` is located when starting Python (an alternative is of course to specify the directory in your `PYTHONPATH`.)

```
>>> from test_sample import Sample
>>> sample = Sample()
>>> print sample.title
None
>>> sample.title = 'Title'
>>> print sample.title
Title
>>> print sample.getDescription()

>>> sample.setDescription('Hello World')
>>> print sample.getDescription()
Hello World
>>> sample.setDescription(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "test_sample.py", line 31, in setDescription
    assert isinstance(value, (str, unicode))
AssertionError
```

As you can see in the last test, non-string object types are not allowed as descriptions and an `AssertionError` is raised.

44.2 Writing the Unit Tests

The goal of writing the unit tests is to convert this informal, manual, and interactive testing session into a formal test class. Python provides already a module called `unittest` for this purpose, which is a port of the Java-based unit testing product, JUnit, by Kent Beck and Erich Gamma. There are three levels to the testing framework (this list deviates a bit from the original definitions as found in the Python library documentation. [1]).

[1] http://www.python.org/doc/current/lib/module-unittest.html

The smallest unit is obviously the “test”, which is a single method in a `TestCase` class that tests the behavior of a small piece of code or a particular aspect of an implementation. The “test case” is then a collection tests that share the same setup/inputs. On top of all of this sits the “test suite” which is a collection of test cases and/or other test suites. Test suites combine tests that should be executed together. With the correct setup (as shown in the example below), you can then execute test suites. For large projects like Zope 3, it is useful to know that there is also the concept of a test runner, which manages the test run of all or a set of tests. The runner provides useful feedback to the application, so that various user interaces can be developed on top of it.

But enough about the theory. In the following example, which you can simply put into the same file as your code above, you will see a test in common Zope 3 style.

```
 1 import unittest
 2 
 3 class SampleTest(unittest.TestCase):
 4     """Test the Sample class"""
 5 
 6     def test_title(self):
 7         sample = Sample()
 8         self.assertEqual(sample.title, None)
 9         sample.title = ’Sample Title’
10         self.assertEqual(sample.title, ’Sample Title’)
11 
12     def test_getDescription(self):
13         sample = Sample()
14         self.assertEqual(sample.getDescription(), ’’)
15         sample._description = "Description"
16         self.assertEqual(sample.getDescription(), ’Description’)
17 
18     def test_setDescription(self):
19         sample = Sample()
20         self.assertEqual(sample._description, ’’)
21         sample.setDescription(’Description’)
22         self.assertEqual(sample._description, ’Description’)
23         sample.setDescription(u’Description2’)
24         self.assertEqual(sample._description, u’Description2’)
25         self.assertRaises(AssertionError, sample.setDescription, None)
26 
27 
28 def test_suite():
29     return unittest.TestSuite((
30         unittest.makeSuite(SampleTest),
31         ))
32 
33 if __name__ == ’__main__’:
34     unittest.main(defaultTest=’test_suite’)
```

- ▷ Line 3–4: We usually develop test classes which must inherit from `TestCase`. While often not done, it is a good idea to give the class a meaningful docstring that describes the purpose of the tests it includes.

- ▷ Line 6, 12 & 18: When a test case is run, a method called `runTests()` is executed. While it is possible to overrride this method to run tests differently, the default option will look for any method whose name starts with `test` and execute it as a single test. This way we can create a “test method” for each aspect, method, function or property of the code to be tested. This default is very sensible and is used everywhere in Zope 3.

Note that there is no docstring for test methods. This is intentional. If a docstring is specified, it is used instead of the method name to identify the test. When specifying a docstring, we have noticed that it is very difficult to identify the test later; therefore the method name is a much better choice.

▷ Line 8, 10, 14, . . . : The `TestCase` class implements a handful of methods that aid you with the testing. Here are some of the most frequently used ones. For a complete list see the standard Python documentation referenced above.

- `assertEqual(first,second[,msg])`
  Checks whether the `first` and `second` value are equal. If the test fails, the `msg` or `None` is returned.
- `assertNotEqual(first,second[,msg])`
  This is simply the opposite to `assertEqual()` by checking for non-equality.
- `assertRaises(exception,callable,...)`
  You expect the `callable` to raise `exception` when executed. After the `callable` you can specify any amount of positional and keyword arguments for the `callable`. If you expect a group of exceptions from the execution, you can make `exception` a tuple of possible exceptions.
- `assert_(expr[,msg])`
  Assert checks whether the specified expression executes correctly. If not, the test fails and `msg` or `None` is returned.
- `failUnlessEqual()`
  This testing method is equivalent to `assertEqual()`.
- `failUnless(expr[,msg])`
  This method is equivalent to `assert_(expr[,msg])`.
- `failif()`
  This is the opposite to `failUnless()`.
- `fail([msg])`
  Fails the running test without any evaluation. This is commonly used when testing various possible execution paths at once and you would like to signify a failure if an improper path was taken.

▷ Line 6–10: This method tests the `title` attribute of the `Sample` class. The first test should be of course that the attribute exists and has the expected initial value (line 8). Then the title attribute is changed and we check whether the value was really stored. This might seem like overkill, but later you might change the title in a way that it uses properties instead. Then it becomes very important to check whether this test still passes.

▷ Line 12–16: First we simply check that `getDescription()` returns the correct default value. Since we do not want to use other API calls like `setDescription()` we set a new value of the description via the implementation-internal `_description` attribute (line 15). This is okay! Unit tests can make use of implementation-specific attributes and methods. Finally we just check that the correct value is returned.

▷ Line 18–25: On line 21–24 it is checked that both regular and unicode strings are set correctly. In the last line of the test we make sure that no other type of objects can be set as a description and that an error is raised.

▷ 28–31: This method returns a test suite that includes all test cases created in this module. It is used by the Zope 3 test runner when it picks up all available tests. You would basically add the line `unittest.makeSuite(TestCaseClass)` for each additional test case.

▷ 33–34: In order to make the test module runnable by itself, you can execute `unittest.main()` when the module is run.

44.3 Running the Tests

You can run the test by simply calling `pythontest_sample.py` from the directory you saved the file in. Here is the result you should see:

```
.
----------------------------------------------------------------------
n 3 tests in 0.001s
```

The three dots represent the three tests that were run. If a test had failed, it would have been reported pointing out the failing test and providing a small traceback.

When using the default Zope 3 test runner, tests will be picked up as long as they follow some conventions.

- The tests must either be in a package or be a module called `tests`.
- If `tests` is a package, then all test modules inside must also have a name starting with `test`, as it is the case with our name `test_sample.py`.
- The test module must be somewhere in the Zope 3 source tree, since the test runner looks only for files there.

In our case, you could simply create a `tests` package in `ZOPE3/src` (do not forget the `__init__.py` file). Then place the `test_sample.py` file into this directory.

You you can use the test runner to run only the sample tests as follows from the Zope 3 root directory:

```
python test.py -vp tests.test_sample
```

The `-v` option stands for verbose mode, so that detailed information about a test failure is provided. The `-p` option enables a progress bar that tells you how many tests out of all have been completed. There are many more options that can be specified. You can get a full list of them with the option `-h`: `pythontest.py-h`.

The output of the call above is as follows:

```
nfiguration file found.
nning UNIT tests at level 1
nning UNIT tests from /opt/zope/Zope3
 3/3 (100.0%): test_title (tests.test_sample.SampleTest)
----------------------------------------------------------------------
n 3 tests in 0.002s
```

```
nning FUNCTIONAL tests at level 1
nning FUNCTIONAL tests from /opt/zope/Zope3

-------------------------------------------------------------------
n 0 tests in 0.000s
```

- ▷ Line 1: The test runner uses a configuration file for some setup. This allows developers to use the test runner for other projects as well. This message simply tells us that the configuration file was found.
- ▷ Line 2–8: The unit tests are run. On line 4 you can see the progress bar.
- ▷ Line 9–15: The functional tests are run, since the default test runner runs both types of tests. Since we do not have any functional tests in the specified module, there are no tests to run. To just run the unit tests, use option `-u` and `-f` for just running the functional tests. See "Writing Functional Tests" for more detials on functional tests.

Exercises

1. It is not very common to do the setup – in our case `sample=Sample()` – in every test method. Instead there exists a method called `setUp()` and its counterpart `tearDown` that are run before and after each test, respectively. Change the test code above, so that it uses the `setUp()` method. In later chapters and the rest of the book we will frequently use this method of setting up tests.

2. Currently the `test_setDescription()` test only verifies that None is not allowed as input value.

   (a) Improve the test, so that all other builtin types are tested as well.

   (b) Also, make sure that any objects inheriting from `str` or `unicode` pass as valid values.