CHAPTER 12

Testing

Testing is the process of ensuring that the code you write performs in the way it was intended. Testing encompasses many different aspects that are relevant to a Pylons application. For example, it is important the libraries do what they are supposed to, that the controllers return the correct HTML pages for the actions they are performing, and that the application behaves in a way that the users find intuitive.

Here are some of the many reasons why writing effective tests is a very good idea when developing a Pylons application:

Fast refactoring: Over the course of most software development projects you'll find that a lot of the code you wrote at the very start of the project is modified before the final release. This isn't necessarily because the code contains bugs; it could be because requirements have changed or because you've spotted a better way of implementing the same functionality. If you have already written tests for the code you need to refactor and the tests pass after you've updated the code, then you can be confident that the changes you made are unlikely to have introduced any unexpected bugs.

Ensuring simple design: If writing unit tests for a particular set of classes, methods, or functions turns out to be a difficult process, it is likely that the code is too complicated and not adequately exposing the API that another developer interacting with your code might eventually need. The fact you are able to write a unit test for a piece of code goes some way to ensuring it is correctly designed.

Use case documentation: A set of tests serves to define the use cases for the code that is being tested. This means the tests also form very effective documentation of how the code should work.

At its heart, testing is about having confidence in the code you have written. If you have written good tests, you can have a good degree of confidence that your code works as it should. Having confidence is especially important if you are using beta or prerelease code in your application. As long as you write tests to ensure the features of the product you are using behave as you require, then you can have a degree of confidence that it is OK to use those particular features in your application.

Although writing effective tests does take time, it is often better to spend time writing tests early in the project than debugging problems later. If you have written tests, every time you make a change to the code, then you can run the tests to see whether any of them fail. This gives you instant feedback about any unforeseen consequences your code change has had. Without the tests, bugs that are introduced might not be picked up for a long time, allowing for the possibility that new code you write might depend on the code working in the incorrect way. If this happens, fixing the bug would then break the new code you had written. This is why later in a project fixing minor bugs can sometimes create major problems. By failing to write effective tests, you can sometimes end up with a system that is difficult to maintain.

It is worth noting that Pylons and all the components that make up Pylons have their own automated test suites. The Pylons tests are run every night on the latest development source using a tool called Buildbot, and the results are published online. Without these extensive test suites, the Pylons developers would not be able to have the confidence in Pylons that they do.

Types of Testing and the Development Process

In this chapter, I'll describe three types of testing you can use to help avoid introducing bugs into your Pylons project during the course of development:

- Unit testing
- Functional testing
- User testing

If you are interested in reading more about other types of software testing, the Wikipedia page is a good place to start: `http://en.wikipedia.org/wiki/Portal:Software_Testing`.

The most common form of testing is *unit testing*. A unit test is a procedure used to validate that individual units of your source code produce the expected output given a known input. In Python, the smallest testable parts of a library or application are typically functions or methods. Unit tests are written from a programmer's perspective to ensure that a particular set of methods or functions successfully perform a set of tasks. In the context of a Pylons application, you would usually write unit tests for any helpers or other libraries you have written. You might also use a unit test to ensure your model classes and methods work correctly. Your Pylons project has a `test_models.py` file in the `tests` directory for precisely this purpose.

While unit tests are designed to ensure individual units of code work properly, *functional tests* ensure that the higher-level code you have written functions in the way that users of your Pylons application would expect. For example, a functional test might ensure that the correct form was displayed when a user visited a particular URL or that when a user clicked a link, a particular entry was added to the database. Functional tests are usually used in the context of a Pylons application to test controller actions.

Some people would argue that the best time to write unit and functional tests is before you have written the code that they would test, and this might be an approach you could take when developing your Pylons application. The advantages of this approach are the following:

- You can be confident the code you have written fulfils your requirements.
- It is likely the code you have written is not overengineered, because you have concentrated on getting the test suite to pass rather than future-proofing your code against possible later changes.
- You know when the code is finished once the test suite passes.

Another approach that helps you write code that meets your requirements without being overengineered is to write the documentation for your Pylons project first and include code samples. Python comes with a library called `doctest` that can analyze the documentation and run the code samples to check that they work in the way you have documented. You'll learn more about `doctest` in Chapter 13.

The final type of testing you should strongly consider integrating into your development process is *user testing*. User testing doesn't involve writing any automated tests at all but instead typically involves getting together a willing and representative group of the intended users of your product and giving them tasks in order to watch how they interact with the system. You then make notes of any tasks they struggle with or any occasions where the software breaks because of their actions and update your Pylons application accordingly, attributing any problems to deficiencies in your software rather than the incompetence of your users.

The end users of your product are often very good people to test your application on because they will have a similar (or greater) knowledge of the business rules for the tasks they are trying to use the system for, but they might not have the technical knowledge you do. This means they are much more likely to do unusual things during the course of their interaction with your application—things you have learned from experience not to do. For example, they might use the Back button after a POST request or copy unusual characters from software such as Microsoft Word that might be in an unexpected encoding. This behavior helps highlight deficiencies in your software that you may not have noticed yourself.

If you are developing a commercial product for a specific set of users, then there is a secondary reason for involving them in the testing of your prototype. It can help familiarize them with the system and give them a chance to highlight any gaps in the software as you are developing it; this in turn vastly reduces the chance of the software not being accepted at the end of the development process because the users have been involved all along the way. Ultimately, if the users of your application are happy with the way your Pylons application works, then it fulfils its goal.

■**Note** Of course, user testing is a topic in its own right, and I won't go into it further here. User testing is also an important part of many software development methodologies that can be used with Pylons.

If you are interested in development methodologies, the Wikipedia articles on agile and iterative development are good places to start and are often useful methodologies to choose for a Pylons project. You might also be interested to read about the Waterfall method, which is a methodology frequently used in larger IT projects but that many people argue often doesn't work in practice.

```
http://en.wikipedia.org/wiki/Agile_software_development
http://en.wikipedia.org/wiki/Iterative_and_incremental_development
http://en.wikipedia.org/wiki/Waterfall_model
```

Unit Testing with nose

Writing unit tests without a testing framework can be a difficult process. The Python community is fortunate to have access to many good quality unit testing libraries including the `unittest` module from the Python standard library, `py.test`, and nose. Pylons uses nose to run the test suite for your Pylons application because it currently has a slightly more advanced feature set than the others. nose is installed automatically when you install Pylons, so it is ready to go.

Before you learn about how to write unit tests specifically for Pylons, let's start by writing some simple tests and using nose to test them.

Introducing nose

Here's what a very simple unit test written using nose might look like:

```
def test_add():
    assert 1+1 == 2

def test_subtract():
    a = 1
    b = a + 1
    c = b-1
    assert b-a == c
```

The example uses the Python keyword `assert` to test whether a particular condition is `True` or `False`. Using the `assert` statement in this way is equivalent to raising an `AssertionError` but is just a little easier to write. You could replace the `test_add()` function with this if you prefer:

```
def test_add():
    if not 1+1 == 2:
        raise AssertionError()
```

nose differentiates between assertion errors and other exceptions. Exceptions as a result of a failed assertion are called *failures*, whereas any other type of exception is treated as an *error*.

To test the example code, save it as `test_maths.py`, and run the following command:

```
$ nosetests test_maths.py
```

You'll see the following output:

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```

For every test that passes, a `.` character is displayed. In this case, since there are only two tests, there are two `.` characters before the summary. Now if you change the line `a = 1` to `a = 3` in the `test_subtract()` function and run the test again, the assertion will fail, so nose tells you about the failure and displays `F` instead of the `.` for the second test:

```
.F
======================================================================
FAIL: test_maths.test_subtract
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/james/lib/python2.5/site-packages/nose-0.10.3-py2.4.egg/➥
nose/case.py", line 182, in runTest
    self.test(*self.arg)
  File "/home/james/Desktop/test_maths.py", line 8, in test_subtract
    assert b-a == c
AssertionError

----------------------------------------------------------------------
Ran 2 tests in 0.002s

FAILED (failures=1)
```

This isn't too helpful because you can't see the values of `a`, `b`, or `c` from the error message, but you can augment this result by adding a message after the `assert` statement to clarify what you are testing like this:

```
assert b-a == c, "The value of b-a does not equal c"
```

which results in the following:

```
.F
======================================================================
FAIL: test_maths.test_subtract
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/james/lib/python2.4/site-packages/nose-0.10.3➥
-py2.5.egg/nose/case.py", line 182, in runTest
    self.test(*self.arg)
  File "/home/james/Desktop/test_maths.py", line 8, in test_subtract
    assert b-a == c
AssertionError: The value of b-a does not equal c

----------------------------------------------------------------------
Ran 2 tests in 0.002s

FAILED (failures=1)
```

This is better but still not particularly helpful because you still can't tell the values of `a`, `b`, or `c` from the output. nose has three different ways to help you solve this problem, covered in the following three sections.

Debug Messages

Any `print` statements you add to your tests are displayed only if the test fails or results in an error. Try modifying the `test_subtract()` function so that it looks like this:

```
def test_subtract():
    a = 3
    print "a is %s"%a
    b = a + 1
    print "b is %s"%b
    c = b-1
    print "c is %s"%c
    assert b-a == c
```

If you run the test again, you'll see the following extra information displayed after the `AssertionError`:

```
-------------------- >> begin captured stdout << ---------------------
a is 3
b is 4
c is 3

--------------------- >> end captured stdout << ----------------------
```

From this you can easily see that `4-3 != 3`, but the test output wouldn't be cluttered with these debug messages unless the test failed. If you would prefer nose to always print debug messages like these, you can use the `-s` option so that it doesn't capture the standard output stream.

Detailed Errors

If you run nose with the `-d` flag, it will try to display the values of the variables in the `assert` statement:

```
$ nosetests -d test_maths.py
```

In this case, the output also contains the following line, so you can immediately see the mistake:

```
>>  assert 4-3 == 3
```

Command-Line Debugging

If you want even more flexibility to debug the output from your tests, you can start nose with the `--pdb` and `--pdb-failures` options, which drop nose into debugging mode if it encounters any errors or failures, respectively. As the option names suggest, nose invokes `pdb` (the Python debugger), so you can use the full range of commands supported by the `pdb` module.

Let's give it a try—start by running the test again with the new flags set:

```
$ nosetests --pdb --pdb-failures test_maths.py
```

Now when the failure occurs, you'll see the `pdb` prompt:

```
.> /home/james/Desktop/test_maths.py(11)test_subtract()
-> assert b-a == c
(Pdb)
```

You can display a list of commands with `h`:

```
Documented commands (type help <topic>):
========================================
EOF    break  condition  disable  help    list  q       step     w
a      bt     cont       down     ignore  n     quit    tbreak   whatis
alias  c      continue   enable   j       next  r       u        where
args   cl     d          exit     jump    p     return  unalias
b      clear  debug      h        l       pp    s       up

Miscellaneous help topics:
==========================
exec  pdb

Undocumented commands:
======================
retval  rv
```

Of these, some of the most important are `l`, which lists the code nearby, and `q`, which exits the debugger so that the tests can continue. The prompt also works a bit like a Python shell, allowing you to enter commands. Here's an example session where you print some variables, obtain help on the `l` command, and then exit the debugger with `q`:

```
(Pdb) print b-a
1
(Pdb) print c
3
(Pdb) h l
l(ist) [first [,last]]
List source code for the current file.
Without arguments, list 11 lines around the current line
or continue the previous listing.
With one argument, list 11 lines starting at that line.
With two arguments, list the given range;
if the second argument is less than the first, it is a count.
(Pdb) l
```

```
  6         print "a is %s"%a
  7         b = a + 1
  8         print "b is %s"%b
  9         c = b-1
 10         print "c is %s"%c
 11  ->     assert b-a == c
 12
[EOF]
(Pdb) q;
```

The `pdb` module and all its options are documented at `http://docs.python.org/lib/module-pdb.html`.

Search Locations

nose uses a set of rules to determine which tests it should run. Its behavior is best described by the text from the nose documentation:

> *nose collects tests automatically from python source files, directories and packages found in its working directory (which defaults to the current working directory). Any python source file, directory or package that matches the* `testMatch` *regular expression (by default:* `(?:^|[\b_\.-])[Tt]est)` *will be collected as a test (or source for collection of tests). In addition, all other packages found in the working directory will be examined for python source files or directories that match* `testMatch`*. Package discovery descends all the way down the tree, so* `package.tests` *and* `package.sub.tests` *and* `package.sub.sub2.tests` *will all be collected.*
>
> *Within a test directory or package, any python source file matching* `testMatch` *will be examined for test cases. Within a test module, functions and classes whose names match* `testMatch` *and* `TestCase` *subclasses with any name will be loaded and executed as tests.*

To specify which tests you want to run, you can pass test names on the command line. Here's an example that will search `dir1` and `dir2` for test cases and will also run the `test_b()` function in the module `test_a.py` in the `tests` directory. All these tests will be looked for in the `some_place` directory instead of the current working directory because the code uses the `-w` flag:

```
$ nosetests -w some_place dir1 dir2 tests/test_a.py:test_b
```

When you are developing a Pylons application, you would normally run `nosetests` from the Pylons project directory (the directory containing the `setup.py` file) so that nose can automatically find your tests.

■ **Note** For more information about nose, see the wiki at `http://code.google.com/p/python-nose/wiki/NoseFeatures`.

Functional Testing

Pylons provides powerful unit testing capabilities for your web application utilizing `paste.fixture` (documented at `http://pythonpaste.org/testing-applications.html#the-tests-themselves`) to emulate requests to your web application. Pylons integrates `paste.fixture` with nose so that you can test Pylons applications using the same techniques you learned for nose in the previous section.

■Note It is likely that at some point Pylons will switch to using the newer WebTest package, but since WebTest is simply an upgrade of `paste.fixture` with some better support for Pylons' newer `request` and `response` objects, the upgrade shouldn't introduce huge changes, and therefore the contents of this section should still largely apply.

To demonstrate functional testing with `paste.fixture`, let's write some tests for the SimpleSite application. If you look at the SimpleSite project, you'll notice the `tests` directory. Within it is a `functional` directory for functional tests that should contain one file for each controller in your application. These are generated automatically when you use the `paster controller` command to add a controller to a Pylons project. To get started, update the `tests/functional/test_page.py` file that was generated when you created the `page` controller so that it looks like this:

```
from simplesite.tests import *

class TestPageController(TestController):

    def test_view(self):
        response = self.app.get(url_for(controller='page', action='view', id=1))
        assert 'Home' in response
```

The page controller doesn't have an `index()` action because you replaced it as part of the tutorial in Chapter 8, so the previous example tests the `view()` action instead.

The `self.app` object is a Web Server Gateway Interface application representing the whole Pylons application, but it is wrapped in a `paste.fixture.TestApp` object (documented at `http://pythonpaste.org/modules/fixture.html`). This means the `self.app` object has the methods `get()`, `post()`, `put()`, `delete()`, `do_request()`, `encode_multipart()`, and `reset()`. Unless you are doing something particularly clever, you would usually just use `get()` and `post()`, which simulate GET and POST requests, respectively.

`get(url, params=None, headers=None, extra_environ=None, status=None, expect_errors=False)`: This gets the URL path specified by `url` using a GET request and returns a response object.

- `params`: A query string, or a dictionary that will be encoded into a query string. You may also include a query string on the `url` argument.
- `headers`: A dictionary of extra headers to send.
- `extra_environ`: A dictionary of environmental variables that should be added to the request.

- `status`: The integer status code you expect (if not 200 or 3xx). If you expect a 404 response, for instance, you must give `status=404`, or an exception will be raised. You can also give a wildcard, like `'3*'` or `'*'`.
- `expect_errors`: If this is not `True`, then if anything is written to `wsgi.errors`, an exception wlll be raised. You'll learn about `wsgi.errors` in Chapters 16 and 20. If the value is set to `True`, then non-200/3xx responses are OK.

`post(url, params='', headers=None, extra_environ=None, status=None, upload_files=None, expect_errors=False)`: This is very similar to the `get()` method, but it performs a POST request, so `params` are put in the body of the request rather than the query string. It takes similar arguments and returns a response object.

- `upload_files`: Should be a list of `[(fieldname, filename, file_content)]` representing files to upload. You can also use just `[(fieldname, filename)]`, and the file content will be read from disk.

The example you've just added to `tests/functional/test_page.py` uses the `get()` method to simulate a GET request to the URL `/page/view/1`. Because a fully configured Pylons environment is set up in the `simplesite.tests` module, you are able to use `url_for()` to generate the URL in the same way you would in a Pylons controller. The `get()` method returns a `paste.fixture response` object. You can then use this to check the response returned was the one you expected. In this case, you check that the home page contains the text `'Home'` somewhere in the response.

In addition to the methods on the `self.app()` object, Pylons also gives you access to some of the Pylons globals that have been created during the request. They are assigned as attributes of the `paste.fixture response` object:

`response.session`: Session object

`response.req`: The Pylons `request` object based on the WebOb `Request`

`response.c`: The template context global containing variables passed to templates

`response.g`: The Pylons app globals object

`response.response`: The Pylons `response` global

To use them, just access the attributes of the response *after* you've used a `get()` or `post()` method:

```
def test_view(self):
    response = self.app.get(url_for(controller='page', action='view', id=1))
    assert 'Home' in response
    assert 'REQUEST_METHOD' in response.req.environ
```

Note The `paste.fixture` response object already has its own request object assigned to it as the `.request` attribute, which is why the Pylons request global is assigned to the `.req` attribute instead.

For simple cases, it is fine to work with the `paste.fixture` response object, but for more complicated cases, you will probably prefer to work with the more familiar Pylons response global available as `response.response`.

Before you test the previous method, you really need to understand exactly how the test setup works because, as it stands, it could damage your development setup. Let's see why in the next section.

How Does the Test Setup Work?

To run the test, you would execute the nosetests command, but before you do, let's consider what this command actually does.

When the nosetests command is executed, it reads some of its configuration from the project's setup.cfg file. This file contains a section that looks like this:

```
[nosetests]
with-pylons=test.ini
```

This tells nose to use the test.ini file to create a Pylons application rather than the development.ini file you've been using so far.

■ **Note** You can also choose the config file to use for the tests by specifying the --with-pylons option on the command line. Likewise, you can also put other nosetests command-line options in the setup.cfg file if that is more convenient. Here are some examples of options you could set:

```
[nosetests]
verbose=True
verbosity=2
with-pylons=test.ini
detailed-errors=1
```

The test.ini file is specifically for your testing configuration and (if properly configured) allows you to keep your testing and development setups completely separate. It looks like this:

```
#
# SimpleSite - Pylons testing environment configuration
#
# The %(here)s variable will be replaced with the parent directory of this file
#
[DEFAULT]
debug = true
# Uncomment and replace with the address which should receive any error reports
#email_to = you@yourdomain.com
smtp_server = localhost
error_email_from = paste@localhost

[server:main]
use = egg:Paste#http
host = 0.0.0.0
port = 5000

[app:main]
use = config:development.ini

# Add additional test specific configuration options as necessary.
```

This should seem very familiar, but notice the line marked in bold in the [app:main] section. This causes the test.ini [app:main] section to use *exactly* the same configuration as the development.ini file's [app:main] section. Although this can save you some effort in some circumstances, it can also cause your tests to interfere with your development setup if you are not careful.

Once nosetests has correctly parsed the test.ini file, it will look for available tests. In doing so, it imports the tests/__init__.py module, which executes this line:

```
SetupCommand('setup-app').run([config['__file__']])
```

Although it looks fairly innocuous, this results in the equivalent of this command being executed:

```
$ paster setup-app test.ini
```

You'll recall from Chapter 8 that this will run the project websetup.py file's setup_app() function with the configuration from test.ini, but because the test.ini file currently uses the configuration from the [app:main] section of the development.ini file, it will be called with the same sqlalchemy.url as your development setup. If your websetup.py was badly written, this could damage the data in your development database.

To make matters worse, the test.ini file doesn't come with the development.ini file's logging configuration, so you can't even see what is happening behind the scenes. Copy all the logging lines from the development.ini file to the end of test.ini starting with # Logging configuration and ending at the end of the file. If you run nosetests again, you will see what has been happening behind the scenes:

```
$ nosetests
20:14:02,807 INFO  [simplesite.websetup] Adding home page...
20:14:02,918 INFO  [simplesite.websetup] Successfully set up.
.
----------------------------------------------------------------------
Ran 1 test in 0.347s

OK
```

As you can see, every time the test suite was run, a new home page was accidentally added to the database. You can confirm this by starting the Paste HTTP server and visiting http://localhost:5000/page/list to verify that an extra home page has been added.

Now that you understand what is happening, update the test.ini file with its own configuration so the [app:main] section looks like this:

```
[app:main]
use = egg:SimpleSite
full_stack = true
cache_dir = %(here)s/data
beaker.session.key = simplesite
beaker.session.secret = somesecret

# SQLAlchemy database URL
sqlalchemy.url = sqlite:///%(here)s/test.db
```

Notice that sqlalchemy.url has been changed to use test.db. This still doesn't prevent a new home page from being added each time the tests run, so you should update websetup.py too. Ideally, you want a completely fresh database each time the tests are run so that they are consistent each time. To achieve this, you need to know which config file is being used to run the tests. The setup_app() function takes a conf object as its second argument. This object has a .filename attribute that contains the name of the file used to invoke the setup_app() function.

Update the setup_app() function in websetup.py to look like this. Notice the import of os.path as well as the code to drop existing tables if using the test.ini file.

```
"""Setup the SimpleSite application"""
import logging
import os.path
from simplesite import model

from simplesite.config.environment import load_environment

log = logging.getLogger(__name__)

def setup_app(command, conf, vars):
    """Place any commands to setup simplesite here"""
    load_environment(conf.global_conf, conf.local_conf)
    # Load the models
    from simplesite.model import meta
    meta.metadata.bind = meta.engine
    filename = os.path.split(conf.filename)[-1]
    if filename == 'test.ini':
        # Permanently drop any existing tables
        log.info("Dropping existing tables...")
        meta.metadata.drop_all(checkfirst=True)
    # Continue as before
    # Create the tables if they aren't there already
    meta.metadata.create_all(checkfirst=True)
    log.info("Adding home page...")
    page = model.Page()
    page.title=u'Homepage'
    page.content = u'Welcome to the SimpleSite home page.'
    meta.Session.save(page)
    meta.Session.commit()
    log.info("Successfully set up.")
```

With these changes in place, let's run the test again:

```
$ nosetests
14:02:58,646 INFO  [simplesite.websetup] Dropping existing tables...
... some lines of output ommitted ...
14:02:59,603 INFO  [simplesite.websetup] Adding homepage...
14:02:59,617 INFO  [sqlalchemy.engine.base.Engine.0x...26ec] BEGIN
14:02:59,622 INFO  [sqlalchemy.engine.base.Engine.0x...26ec] INSERT INTO page ➥
(content, posted, title, heading) VALUES (?, ?, ?, ?)
14:02:59,623 INFO  [sqlalchemy.engine.base.Engine.0x...26ec] [u'Welcome to ➥
the SimpleSite home page.', '2008-11-04 14:02:59.622472', u'Home Page', None]
14:02:59,629 INFO  [sqlalchemy.engine.base.Engine.0x...26ec] COMMIT
14:02:59,775 INFO  [simplesite.websetup] Successfully set up
.
----------------------------------------------------------------------
Ran 1 test in 0.814s

OK
```

This time the `setup_app()` function can determine that it is being called with the `test.ini` config setup, so it drops all the tables before performing the normal setup. Because you updated the `test.ini` file with a new SQLite database, you'll notice the test database `test.db` has been created in the same directory as `test.ini`.

Testing the save() Action

The page controller's `save()` action currently looks like this:

```
@restrict('POST')
@validate(schema=NewPageForm(), form='edit')
def save(self, id=None):
    page_q = meta.Session.query(model.Page)
    page = page_q.filter_by(id=id).first()
    if page is None:
        abort(404)
    for k,v in self.form_result.items():
        if getattr(page, k) != v:
            setattr(page, k, v)
    meta.Session.commit()
    session['flash'] = 'Page successfully updated.'
    session.save()
    # Issue an HTTP redirect
    response.status_int = 302
    response.headers['location'] = h.url_for(controller='page', action='view',
        id=page.id)
    return "Moved temporarily"
```

Let's write a test to check that this action behaves in the correct way:

- GET requests are disallowed by the `@restrict` decorator.
- The action returns a 404 Not Found response if no ID is specified.
- The action returns a 404 Not Found response for IDs that don't exist.
- Invalid data should result in the form being displayed.
- The action saves the updated data in the database.
- The action sets a flash message in the session.
- The action returns a redirect response to redirect to the create action.

You could write the tests for each of these in a single method of the `TestPageController` class, but if one of the tests failed for any reason, nose would not continue with the rest of the method. On the other hand, some of the tests are dependent on other tests passing, so you cannot write them all as separate methods either. For example, you can't test whether a flash message was set if saving the page failed. To set up the tests correctly, let's create the following methods:

```
def test_save_prohibit_get(self):
    """Tests to ensure that GET requests are prohibited"""

def test_save_404_invalid_id(self):
    """Tests that a 404 response is returned if no ID is specified
    or if the ID doesn't exist"""

def test_save_invalid_form_data(self):
    """Tests that invalid data results in the form being returned with
    error messages"""

def test_save(self):
    """Tests that valid data is saved to the database, that the response redirects
    to the view() action and that a flash message is set in the session"""
```

These tests will require some imports, so add the following lines to the top of `tests/functional/test_page.py`:

```
from routes import url_for
from simplesite.model import meta
from urlparse import urlparse
```

Now let's implement the test methods starting with `test_save_prohibit_get()`, which looks like this:

```
class TestPageController(TestController):

    def test_save_prohibit_get(self):
        """Tests to ensure that GET requests are prohibited"""
        response = self.app.get(
            url=url_for(controller='page', action='save', id='1'),
            params={
                'heading': u'Updated Heading',
                'title': u'Updated Title',
                'content': u'Updated Content',
            },
            status = 405
        )
```

As you can see, the example uses the `get()` method of `self.app` to simulate a GET request to the `save()` action with some sample `params`, which will be sent as part of the query string. By default, the `get()` and `post()` methods expect either a 200 response or a response in the 300s and will consider anything else an error. In this case, you expect the request to be denied with a 405 Method Not Allowed response, so to prevent `paste.fixture` from raising an exception, you have to specify the `status` parameter explicitly. Because `paste.fixture` checks that the status will be 405, you don't have to add another check on the `response` object.

Now let's look at the `test_save_404_invalid_id()` method:

```
  def test_save_404_invalid_id(self):
        ""Tests that a 404 response is returned if no ID is specified
        or if the ID doesn't exist"""
        response = self.app.post(
            url=url_for(controller='page', action='save', id=''),
            params={
               'heading': u'Updated Heading',
               'title': u'Updated Title',
               'content': u'Updated Content',
            },
            status=404
        )
        response = self.app.post(
            url=url_for(controller='page', action='save', id='2'),
            params={
                'heading': u'Updated Heading',
                'title': u'Updated Title',
                'content': u'Updated Content',
            },
            status=404
        )
```

As you can see, this code is similar but uses the `post()` method and performs two tests rather than one. In the first, no ID is specified, and in the second the ID specified doesn't exist. In both cases, you expect a 404 HTTP response, so the `status` parameter is set to `404`.

The `test_save_invalid_form_data()` method is more interesting. Once again a POST request is triggered, but this time the `title` is empty, so the `@validate` decorator should cause the page to be redisplayed with the error message `Please enter a value`:

```
def test_save_invalid_form_data(self):
    """Tests that invalid data results in the form being returned with
    error messages"""
    response = self.app.post(
        url=url_for(controller='page', action='save', id='1'),
        params={
            'heading': u'Updated Heading',
            # title is required so this next entry is invalid
            'title': u'',
            'content': u'Updated Content',
        }
    )
    assert 'Please enter a value' in response
```

As you can see from the last line, the presence of the error message in the response is tested. Because you expect a 200 HTTP response, there is no need to specify the `status` argument, but you can if you like.

Finally, let's look at the `test_save()` method:

```
def test_save(self):
    """Tests that valid data is saved to the database, that the response redirects
    to the view() action and that a flash message is set in the session"""

    response = self.app.post(
        url=url_for(controller='page', action='save', id='1'),
        params={
            'heading': u'Updated Heading',
            'title': u'Updated Title',
            'content': u'Updated Content',
        }
    )

    # Test the data is saved in the database (we use the engine API to
    # ensure that all the data really has been saved and isn't being returned
    # from the session)
    connection = meta.engine.connect()
    result = connection.execute(
        """
        SELECT heading, title, content
        FROM page
        WHERE id=?
        """,
        (1,)
    )
    connection.close()
    row = result.fetchone()
    assert row.heading == u'Updated Heading'
    assert row.title == u'Updated Title'
    assert row.content == u'Updated Content'

    # Test the flash message is set in the session
    assert response.session['flash'] == 'Page successfully updated.'
```

```
        # Check the respone will redirect to the view action
        assert urlparse(response.response.location).path == url_for(
            controller='page', action='view', id=1)
        assert response.status == 302
```

The first part of this test generates a `paste.fixture response` after posting some valid data to the `save()` action. A SQLAlchemy `connection` object is then created to perform a SQL `SELECT` operation directly on the database to check the data really has been updated. Next you check the session contains the flash message. You'll remember from earlier in the chapter that certain Pylons globals including `session` are available as attributes of the `response` object. In this example, `response.session` is tested to ensure the flash message is present. Finally, you want to check the HTTP response headers contain the `Location` header with the correct URL to redirect the browser to the `view()` action. Here we are using the Pylons `response` object because it has a `.location` attribute specifying the location header rather than the `paste.fixture response` object. The location header contains the whole URL, so you use `urlparse()` to just compare that the path component matches the path to the `view()` action.

Once you've implemented the tests, you can check they pass by running `nosetests` in your main project directory:

```
$ nosetests simplesite/tests/functional/test_page.py
... log output omitted ...
..
----------------------------------------------------------------------
Ran 4 tests in 0.391s

OK
```

The tests all pass successfully, so you can be confident the `save()` action functions as it is supposed to function.

■ **Tip** The `TestPageController` is derived from the `TestController` class, which itself subclasses the standard Python `unittest.TestCase` class. This means you can also use its helper methods in your tests. The `unittest.TestCase` object is documented at `http://docs.python.org/lib/testcase-objects.html`. This is well worth a read if you plan to write anything more than simple tests.

Testing Your Own Objects

As you saw earlier in the chapter, Pylons adds certain objects to the `response` object returned by `paste.fixture` when you call the `self.app` object with one of the HTTP methods such as `get()` or `post()`. You can also set up your own objects to be added to the `response` object. If a test is being run, Pylons makes available a `paste.testing_variables` dictionary in the `request.environ` dictionary. Any objects you add to this dictionary are automatically added as attributes to the `paste.fixture response` object. For example, if you had a custom `Cache` object that you wanted to make available in the tests, you might modify the `__call__()` method in the `BaseController` in your project's `lib/base.py` file to look like this:

```
class BaseController(WSGIController):

    def __call__(self, environ, start_response):
        # Add the custom cache object
        if 'paste.testing_variables' in environ:
            environ['paste.testing_variables']['cache'] = CustomCacheObj()
        try:
            return WSGIController.__call__(self, environ, start_response)
        finally:
            meta.Session.remove()
```

In the `TestPageController` you would now find the `response` object has a `.cache` attribute:

```
def test_cache(self):
    response = self.app.get(url(controller='page', action='view', id='1'))
    assert hasattr(response, 'cache') is True
```

For more details on running tests using `paste.fixture`, visit `http://pythonpaste.org/testing-applications.html#the-tests-themselves`.

Interactive Shell

Sometimes it is useful to be able to test your application from the command line. As you saw earlier in the chapter, one method for doing this is to use the `--pdb` and `--pdb-failures` options with nose to debug a failing test, but what if you want to quickly see how a particular part of your Pylons application behaves to help you work out how you should write your test? In that case, you might find the Pylons interactive shell useful.

The Pylons interactive shell enables you to use all the tools you would usually use in your tests but in an interactive way. This is also particularly useful if you can't understand why a particular test is giving you the result it is. The following command starts the interactive shell with the test setup, but you could equally well specify `development.ini` if you wanted to test your development setup:

```
$ paster shell test.ini
```

Here's the output you receive:

```
Pylons Interactive Shell
Python 2.5.1 (r251:54863, Apr 15 2008, 22:57:26)
[GCC 4.0.1 (Apple Inc. build 5465)]

  All objects from simplesite.lib.base are available
  Additional Objects:
  mapper     -  Routes mapper object
  wsgiapp    -  This project's WSGI App instance
  app        -  paste.fixture wrapped around wsgiapp

>>>
```

As you can see, the shell provides access to the same objects as you have access to in the actions of your functional test classes.

You can use the Pylons interactive shell in the same way you would usually use a Python shell. Here are some examples of its use:

```
>>> response = app.get('/page/view/1')
13:24:31,824 INFO  [sqlalchemy.engine.base.Engine.0x..90] BEGIN
13:24:31,828 INFO  [sqlalchemy.engine.base.Engine.0x..90] ➥
SELECT page.id AS page_id, page.content AS page_content, ➥
page.posted AS page_posted, page.title AS page_title,➥
page.heading AS page_heading
FROM page
WHERE page.id = ?
 LIMIT 1 OFFSET 0
13:24:31,828 INFO  [sqlalchemy.engine.base.Engine.0x..90] [1]
>>> assert 'Updated Content' in response
>>> print response.req.environ.has_key('REMOTE_USER')
False
>>>
```

Notice that you receive the same logging output because of the logging configuration you added to `test.ini` earlier in the chapter. Also notice that because you've already run the `nosetests` command, the database currently has the text `Updated Content` for the content rather than the message `Welcome to the SimpleSite home page.`, which was the original value.

Summary

In this chapter, you saw how nose, `paste.fixture`, and the Pylons interactive shell work together to allow you to test Pylons applications. You've seen how to use some of the more common options available to nose to customize the output from the tests and how to debug failures and errors with Python's `pdb` module. You also learned the difference between unit testing, functional testing, and user testing and saw why all of them are important. You also now know exactly how Pylons sets up your tests so that you can customize their behavior by changing the `websetup.py` file or adding new objects to the `paste.fixture` `response` object.

In the next chapter, you'll look at some of the recommended ways to document a Pylons project, and you'll learn about one more type of testing known as a *doctest*, which allows examples from within the documentation to be tested directly.