CHAPTER 2

Getting Started

In the previous chapter, you learned how to set up the environment for Python 3 for Linux, MacOS, and Windows computers. You also looked at a few popular IDEs for Python. In this chapter, we will get started with concepts of test automation. Then we will explore a light and easy way to learn the test automation framework in Python 3, called `doctest`.

A Brief Introduction to Software Testing Concepts

The textbook definition of software testing says that software testing means executing a program or application to find any bugs. Usually, there are multiple stakeholders in the process of software testing. The stakeholders include testers, the management team, consultants, business, customers, and end users. With medium to large-scale projects, software testing is compulsorily done to determine if the software behaves as intended under various sets of inputs and conditions.

Unit Testing

Unit testing is a software testing method in which individual components of the program, called *units,* are tested independently with all the required dependencies. Unit testing is mostly done by the actual programmers, who write the programs for the units. In smaller projects, it is done informally. In most of the very large-scale projects, unit testing is part of a formal process of development with proper documentation and proper schedule/efforts allocated to it.

Test Automation

Test automation is the automated execution and reporting of the outcome of test scenarios and cases. In most large and complex projects, many phases of the testing process are automated. Sometimes the effort of automating the tests is so huge that there is a separate project for automation with a separate team dedicated to it, including a separate reporting structure with separate management. There are several areas and phases of testing that can be automated. Various tools like code libraries and third-party APIs are used for unit testing. Sometimes, the code for unit testing is also generated in an automated way. Unit testing is a prime candidate for automation.

A. Pajankar, *Python Unit Test Automation*, DOI 10.1007/978-1-4842-2677-3_2

The Benefits of Automated Unit Testing

There are many reasons to automate unit tests. Let's consider them one by one.

- Time and effort

  As your codebase grows, the number of modules to be unit tested grows. Manual testing occupies a lot of days of the typical programmer's calendar. To reduce manual testing efforts, you can automate test cases, which then can be automated easily and quickly.

- Accuracy

  Test case execution is a rote and boring activity. Humans can make mistakes. However, an automated test suite will run and return correct results every time.

- Early bug reporting

  Automating unit test cases gives you the distinct advantage of early reporting of bugs and errors. When the automated test suites are run by the scheduler, once the code freezes due to an error, all the logical bugs in the code are quickly discovered and reported, without much human intervention needed.

- Built-in support for unit testing

  There are many programming languages that provide built-in support for writing unit tests by means of libraries dedicated to unit testing. Examples include Python, Java, and PHP.

Using Docstrings

The focus of this chapter is on getting you started with unit test automation in Python. Let's get started with the concept of docstrings and their implementation in Python. Docstrings are going to be immensely useful to you while learning `doctest`.

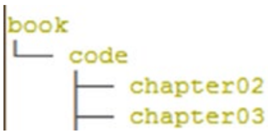
A *docstring* is a string literal that's specified in the source code of a module. It is used to document a specific segment of the code. Code comments are also used for documenting the source code. However, there is a major difference between a docstring and a comment. When the source code is parsed, the comments are not included in the parsing tree as part of the code, whereas docstrings are included in the parsed code tree.

The major advantage of this is that the docstrings are available for use at runtime. Using the functionalities specific to the programming language, you can retrieve the docstring specific to a module. Docstrings are always retained through the entire runtime of the module instance.

Example of a Docstring in Python

Let's see how the concept of the docstring is implemented in Python. A Python docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. A docstring becomes the `__doc__` special attribute of that object.

Let's take a look at a code example of a Python docstring. From this chapter onward, you will be programming quite a lot. I recommend that you create a directory on your computer and create chapter-specific subdirectories within it. As I mentioned earlier, I am using a Linux OS. (My favorite computer, a Raspberry Pi 3 Model B.) I have created a directory called `book` and a directory called `code` under that. The `code` directory has chapter-specific directories containing the code of each chapter. Figure 2-1 shows a graphical representation of the directory structure in form of a tree diagram.



```
book
└── code
    ├── chapter02
    ├── chapter03
```

Figure 2-1. *The suggested directory structure for the book*

Create chapter-specific subdirectories under the directory `code`, as shown in the tree diagram in Figure 2-1. We will use the directory `chapter02` for this chapter, `chapter03` for the next chapter, and so on. Navigate to the `chapter02` directory and save the following code (Listing 2-1) as `test_module01.py` in that directory.

Listing 2-1. test_module01.py

```
"""
This is test_module01.
This is example of multiline docstring.
"""


class TestClass01:
    """This is TestClass01."""

    def test_case01(self):
        """This is test_case01()."""

def test_function01():
    """This is test_function01()."""
```

In (Listing 2-1), we have a test file called `test_module01.py`, which includes `TestClass01` and `test_function01()`. `TestClass01` has a method called `test_case01()`. We have a docstring for all the code units here. The first docstring is a multiline docstring. The rest are examples of single-line docstrings.

Let's see how the docstrings work using the code in Listing 2-1 and an interactive Python session.

Navigate to the chapter02 directory and type python3 to invoke Python 3 in interpreter mode.

```
pi@raspberrypi:~/book/code/chapter02 $ pwd
/home/pi/book/code/chapter02
pi@raspberrypi:~/book/code/chapter02 $ python3
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Import the test module you just created with the following statement:

```
>>> import test_module01
```

You can use the help() function to see the docstrings of the module and its members, as follows.

```
>>> help(test_module01)
```

The output is as follows:

```
Help on module test_module01:


NAME
    test_module01

DESCRIPTION
    This is test_module01.
    This is example of multiline docstring.

CLASSES
    builtins.object
        TestClass01

    class TestClass01(builtins.object)
     |  This is TestClass01.
     |
     |  Methods defined here:
     |
     |  test_case01(self)
     |      This is test_case01().
     |
     |  -------------------------------------------------------------------
    |  Data descriptors defined here:
    |
    |  __dict__
    |      dictionary for instance variables (if defined)
    |
    |  __weakref__
    |      list of weak references to the object (if defined)

FUNCTIONS
    test_function01()
        This is test_function01().

FILE
    /home/pi/book/code/chapter02/test_module01.py
```

You can see the docstring of the individual members using `help()`. Run the following statements and see the output for yourself.

```
>>> help(test_module01.TestClass01)
>>> help(test_module01.TestClass01.test_case01)
>>> help(test_module01.test_function01)
```

As mentioned earlier, a docstring becomes the __doc__ special attribute of that object. You can also use the `print()` function to see the docstring of a module and its members. The following interactive Python session demonstrates that.

```
>>> import test_module01
>>> print(test_module01.__doc__)

This is test_module01.
This is example of multiline docstring.

>>> print(test_module01.TestClass01.__doc__)
This is TestClass01.
>>> print(test_module01.TestClass01.test_case01.__doc__)
This is test_case01().
>>> print(test_module01.test_function01.__doc__)
This is test_function01().
>>>
```

You can find detailed information about the Python docstring on the following PEP pages.

https://www.python.org/dev/peps/pep-0256
https://www.python.org/dev/peps/pep-0257
https://www.python.org/dev/peps/pep-0258

In the next section, you will learn to use docstrings to write simple test cases and execute them with `doctest`.

A Brief Introduction to doctest

`doctest` is the lightweight unit testing framework in Python that uses docstrings to test automation. The `doctest` is packaged with the Python interpreter, so you do not have to install anything separately to use it. It is part of Python's standard library and adheres to Python's "batteries-included" philosophy.

■ **Note** If you're interested, you can read Python's batteries-included philosophy on the PEP 206 page (`https://www.python.org/dev/peps/pep-0206`).

The code in Listing 2-2 is a simple example of a test module with two functions and two tests for each function.

Listing 2-2. test_module02.py

```
"""
Sample doctest test module...
test_module02
"""

def mul(a, b):
        """
>>> mul(2, 3)
        6
>>> mul('a', 2)
        'aa'
        """
        return a * b

def add(a, b):
        """
>>> add(2, 3)
        5
>>> add('a', 'b')
        'ab'
        """
        return a + b
```

In Listing 2-2, the test cases are mentioned as the docstrings for the modules and there is nothing specifically calling the `doctest` in the code itself. When the program is executed as a Python 3 program using the command `python3 test,_module02.py` does not produce any output at the command line. In order to see `doctest` in action, you have to run it using the following command at the command prompt:

```
python3 -m doctest -v test_module02.py
```

The output will be as follows,

```
Trying:
    add(2, 3)
Expecting:
    5
ok
Trying:
    add('a', 'b')
Expecting:
    'ab'
ok
Trying:
    mul(2, 3)
Expecting:
    6
ok
Trying:
    mul('a', 2)
Expecting:
    'aa'
ok
1 items had no tests:
    test_module02
2 items passed all tests:
   2 tests in test_module02.add
   2 tests in test_module02.mul
4 tests in 3 items.
4 passed and 0 failed.
Test passed.
```

Let's take a look at how the `doctest` works. By comparing the code—specifically the commands for execution and output—you can figure out quite a few things. `doctest` works by parsing docstrings. Whenever `doctest` finds an interactive Python prompt in the `doctest` documentation of a module, it treats its output as the expected output. Then it runs the module and its members by referring to the docstrings. It compares the actual output against the output specified in the docstrings. Then it marks the test pass or fail. You have to use `-m doctest` while executing the module to let the interpreter know that you need to use the `doctest` module to execute the code.

The command-line argument `-v` stands for *verbose* mode. You must use it because, without it, the test will not produce any output unless it fails. Using verbose produces an execution log irrespective of whether the test passes or fails.

Failing Tests

In Listing 2-2, all the tests passed with no hassles. Now, let's see how a test fails. In Listing 2-2, replace + on the last line of the code with an * (asterisk) and run the test again. You will find the following output:

```
Trying:
    add(2, 3)
Expecting:
    5
**********************************************************************
File "/home/pi/book/code/chapter02/test_module02.py", line 19, in
test_module02.add
Failed example:
    add(2, 3)
Expected:
    5
Got:
    6
Trying:
    add('a', 'b')
Expecting:
    'ab'
**********************************************************************
File "/home/pi/book/code/chapter02/test_module02.py", line 21, in
test_module02.add
Failed example:
    add('a', 'b')
Exception raised:
    Traceback (most recent call last):
      File "/usr/lib/python3.4/doctest.py", line 1324, in __run
        compileflags, 1), test.globs)
      File "<doctest test_module02.add[1]>", line 1, in <module>
        add('a', 'b')
      File "/home/pi/book/code/chapter02/test_module02.py", line 24, in add
        return a * b
    TypeError: can't multiply sequence by non-int of type 'str'
Trying:
    mul(2, 3)
Expecting:
    6
ok
Trying:
    mul('a', 2)
Expecting:
    'aa'
```

```
ok
1 items had no tests:
    test_module02
1 items passed all tests:
   2 tests in test_module02.mul
**********************************************************************
1 items had failures:
   2 of   2 in test_module02.add
4 tests in 3 items.
2 passed and 2 failed.
***Test Failed*** 2 failures.
```

You can clearly see two failures in the execution log. The tests usually fail due to one or more of the following reasons:

- Faulty logic in the code
- Faulty input into the code
- Faulty test case

In this case, there are two failures in the test. The first one is due to faulty logic. The second failure is due to faulty logic in the code and the wrong type of input given to the function to be tested.

Correct the code by replacing the * in the last line with +. Then change the line that has `'aa'` to aa and run the test again. This will demonstrate the third cause of test failure (a faulty test case).

Separate Test File

You can also write your tests in a separate test file and run them separately from the code to be tested. This helps maintain the test modules/code separately from the development code. Create a file called `test_module03.txt` in the same directory and add the code shown in Listing 2-3 to it.

Listing 2-3. test_module03.txt

```
>>> from test_module02 import *
>>> mul(2, 3)
6
>>> mul('a', 2)
'aa'
>>> add(2, 3)
5
>>> add('a', 'b')
'ab'
```

You can run this test in the usual way, by running the following command in the command prompt:

```
python3 -m doctest -v test_module03.txt
```

The output will be as follows:

```
Trying:
    from test_module02 import *
Expecting nothing
ok
Trying:
    mul(2, 3)
Expecting:
    6
ok
Trying:
    mul('a', 2)
Expecting:
    'aa'
ok
Trying:
    add(2, 3)
Expecting:
    5
ok
Trying:
    add('a', 'b')
Expecting:
    'ab'
ok
1 items passed all tests:
   5 tests in test_module03.txt
5 tests in 1 items.
5 passed and 0 failed.
Test passed.
```

Advantages and Disadvantages of doctest

As you have learned, `doctest` is a very simple and intuitive framework for novice-level testing in Python. It does not require any installation and you can quickly get started with it without needing to know any API. It is mostly used for the following purposes:

- To verify if the code documentation is up to date and the interactive examples in the docstring still work after making changes to the code.
- To perform module-wise basic regression testing.

- To write illustrative tutorials and documentation that doubles as the test case for the package and module.

However, doctest has its own set of limitations. It does not have true API for testing. Also doctest tests tend to be static in nature and cannot be parameterized.

Readers are advised to visit the doctest documentation page at https://docs.python.org/3/library/doctest.html for detailed usage and more examples.

Conclusion

In this chapter, you learned the basics of software testing. You explored a light testing framework, called doctest. It's a good module for simple projects for novice Python users. However, due to its lack of advanced features like testrunner, test discovery, and test fixtures, doctest is not used in large projects. In the next chapter, we will discuss a built-in xUnit style test automation framework for Python, called unittest.

CHAPTER 3

Unittest

The last chapter discussed of the concepts of test automation. You learned about docstring and doctest and their use in writing simple, static, yet elegant test cases for Python 3 programs. However, due to the lack of features like API, configurable tests, and test fixtures, doctest enjoys very limited popularity. You need to explore a powerful API library for automating complex real-life projects and learning Python's built-in unittest module is your first step toward it. This is a detailed and long chapter. You will learn many new concepts like test fixtures, automated test discovery, organizing your codebase, etc. in this chapter. You will use these concepts throughout the book and see their implementation in various more advanced test automation libraries in Python. So, I recommend that you follow every topic in this chapter very carefully.

unittest came to life as a third-party module PyUnit. PyUnit was the Python port for JUnit. JUnit is Java's xUnit-style unit test automation framework.

The PyUnit became part of the Python Standard library from version 2.5 onward. It was rechristened unittest. Unittest is the *batteries-included* test automation library of Python, which means you do not have to install an additional library or tool in order to start using it. Anyone who is familiar with xUnit-style libraries in other programming languages (such as JUnit for Java, PHPUnit for PHP, CPPUnit for C++, etc.) will find it very easy to learn and use unittest.

Introduction to xUnit

Let's take a look at the xUnit philosophy in brief. xUnit is the collective name for several unit testing frameworks for various languages. All the xUnit-style unit testing frameworks more or less derive their functionality, structure, and coding style from Smalltalk's unit testing framework SUnit. Kent Beck designed and wrote SUnit. After it gained popularity, it was ported to Java as JUnit by Kent Beck and Erich Gamma. Eventually, it was ported to almost every programming language. Now most of the programming languages come pre-packaged with at least one xUnit-style test automation library. Also, many programming languages like Python and Java have more than one xUnit-style framework. Java has TestNG in addition to JUnit. Python has nose, pytest, and Nose2 apart from unittest.

A. Pajankar, *Python Unit Test Automation*, DOI 10.1007/978-1-4842-2677-3_3

All the xUnit-style test automation libraries follow a common architecture. The following are the major components of the architecture:

- *Test case class*: This is the base class of all the test classes in the test modules. All the test classes are derived from here.
- *Test fixtures*: These are functions or methods that run before and after blocks of the test code execute.
- *Assertions*: These functions or methods are used to check the behavior of the component being tested. Most of the xUnit-style frameworks are packed with powerful assertion methods.
- *Test suite*: This is the collection or group of related tests that can be executed or scheduled to be executed together.
- *Test runner*: This is the program or block of code that runs the test suite.
- *Test result formatter*: This formats the test results to produce the output of test execution in various human readable formats like plaintext, HTML, and XML.

The implementation details of these components of xUnit differ slightly across the unit testing frameworks. Interestingly, this enables programmers to choose the framework based on the needs of their projects and their comfort.

If you are a seasoned programmer who has experience with any of these frameworks, you will be quickly able to translate your knowledge to Python code. If you do not have prior experience with any of the xUnit-style frameworks, then after reading the book, executing all the examples in the book, and solving all the exercises, you will be able to get started with any of the xUnit frameworks on your own without much hand-holding.

Using Unittest

This section starts with unittest. It begins with the most fundamental concept of a test class.

For this chapter, create a directory called chapter03 in the code directory. In chapter03, create another directory called test (you will learn later in the chapter why you need that additional directory). Save the code in Listing 3-1 as test_module01.py.

Listing 3-1. test_module01.py

```
import unittest

class TestClass01(unittest.TestCase):

        def test_case01(self):
                my_str = "ASHWIN"
                my_int = 999
                self.assertTrue(isinstance(my_str, str))
                self.assertTrue(isinstance(my_int, int))
        def test_case02(self):
                my_pi = 3.14
                self.assertFalse(isinstance(my_pi, int))

if __name__ == '__main__':
        unittest.main()
```

In the code in Listing 3-1, the `import unittest` statement imports the `unittest` module. `TestClass01` is the test class. It is subclassed from the `TestCase` class in the `unittest` module. The class methods `test_case01()` and `test_case02()` are test methods, as their names start with `test_` (You will learn about the guidelines and naming conventions for writing tests later in the chapter.) The `assertTrue()` and `assertFalse()`methods are assertion methods which check if the argument passed to them is `True` or `False`, respectively. If the argument meets the `assert` condition, the test case passes; otherwise, it fails. `unittest.main()` is the test runner. We will explore more `assert` methods in detail later.

Navigate to the test directory as follows:

```
cd ~/book/code/chapter03/test
```

Run the following command:

```
python3 test_module01.py
```

It yields the following output:

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.002s

OK
```

It says OK, as both the tests passed. This is one of the ways you can write and execute tests.

The test execution did not display much information. That's because verbosity is disabled by default. You can run the tests in verbose mode using the `-v` command-line option. Run the following command at the command prompt:

```
python3 test_module01.py -v
```

The verbose output is as follows:

```
test_case01 (__main__.TestClass01) ... ok
test_case02 (__main__.TestClass01) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.004s

OK
```

Certainly, the verbose execution mode provides more insight about the test execution. We will be using this mode very frequently throughout the book for running the tests and gathering the log for test executions.

Order of Execution of the Test Methods

Now, you will see the order in which the test methods are executed. Check the code in Listing 3-2.

Listing 3-2. test_module02.py

```
import unittest
import inspect

class TestClass02(unittest.TestCase):

        def test_case02(self):
                print("\nRunning Test Method : " + inspect.stack()[0][3])

        def test_case01(self):
                print("\nRunning Test Method : " + inspect.stack()[0][3])

if __name__ == '__main__':
        unittest.main(verbosity=2)
```

In the code in Listing 3-2, `inspect.stack()[0][3]` method prints the name of the current test method. It's useful for debugging when you want to know the order that the methods are executed in the test class. The output of the code in Listing 3-2 is as follows:

```
test_case01 (__main__.TestClass02) ...
Running Test Method : test_case01
ok
test_case02 (__main__.TestClass02) ...
Running Test Method : test_case02
ok

----------------------------------------------------------------------
Ran 2 tests in 0.090s

OK
```

Note that the test methods ran in alphabetical order, irrespective of the order of the test methods in the code.

Verbosity Control

In earlier examples, you controlled the verbosity of test execution through the command while invoking the Python test script in the OS console. Now, you will learn how to control the verbose mode from the code itself. See the code in Listing 3-3 for an example.

Listing 3-3. test_module03.py

```
import unittest
import inspect

def add(x, y):
        print("We're in custom made function : " + inspect.stack()[0][3])
        return(x + y)

class TestClass03(unittest.TestCase):

        def test_case01(self):
                print("\nRunning Test Method : " + inspect.stack()[0][3])
                self.assertEqual(add(2, 3), 5)

        def test_case02(self):
                print("\nRunning Test Method : " + inspect.stack()[0][3])
                my_var = 3.14
                self.assertTrue(isinstance(my_var, float))

        def test_case03(self):
                print("\nRunning Test Method : " + inspect.stack()[0][3])
                self.assertEqual(add(2, 2), 5)

        def test_case04(self):
                print("\nRunning Test Method : " + inspect.stack()[0][3])
                my_var = 3.14
                self.assertTrue(isinstance(my_var, int))

if __name__ == '__main__':
        unittest.main(verbosity=2)
```

In Listing 3-3, you are testing a custom function called `add()` with the `assertEqual()` method. `assertEqual()` takes two arguments and determines if both arguments are equal. If both arguments are equal, the test case passes; otherwise, it fails. We have also written a function called `add()` in the same test module that's not a member of the test class. With `test_case01()` and `test_case03()`, we are testing the correctness of the function.

We are also setting the verbosity to the value `2` in the `unittest.main()` statement. Run the code in Listing 3-3 with the following command:

```
python3 test_module03.py
```

The output is as follows:

```
test_case01 (__main__.TestClass03) ...
Running Test Method : test_case01
We're in custom made function : add
ok
test_case02 (__main__.TestClass03) ...
Running Test Method : test_case02
ok
test_case03 (__main__.TestClass03) ...
Running Test Method : test_case03
We're in custom made function : add
FAIL
test_case04 (__main__.TestClass03) ...
Running Test Method : test_case04
FAIL

======================================================================
FAIL: test_case03 (__main__.TestClass03)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_module03.py", line 23, in test_case03
    self.assertEqual(add(2, 2), 5)
AssertionError: 4 != 5

======================================================================
FAIL: test_case04 (__main__.TestClass03)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_module03.py", line 28, in test_case04
    self.assertTrue(isinstance(my_var, int))
AssertionError: False is not true

----------------------------------------------------------------------
Ran 4 tests in 0.112s

FAILED (failures=2)
```

The test cases `test_case03()` and `test_case04()` failed because the `assert` conditions failed. You now have more information related to the test case failure, since verbosity was enabled in the code.

Multiple Test Classes Within the Same Test File/Module

Until now, the examples included a single test class in a single test file. A `.py` file that contains the test class is also called a *test module*. Now you will see an example (Listing 3-4) of a test module that has multiple test classes.

Listing 3-4. test_module04.py

```
import unittest
import inspect

class TestClass04(unittest.TestCase):

        def test_case01(self):
                print("\nClassname : " + self.__class__.__name__)
                print("Running Test Method : " + inspect.stack()[0][3])

class TestClass05(unittest.TestCase):

        def test_case01(self):
                print("\nClassname : " + self.__class__.__name__)
                print("Running Test Method : " + inspect.stack()[0][3])

if __name__ == '__main__':
        unittest.main(verbosity=2)
```

The following is the output after running the code in Listing 3-4:

```
test_case01 (__main__.TestClass04) ...
Classname : TestClass04
Running Test Method : test_case01
ok
test_case01 (__main__.TestClass05) ...
Classname : TestClass05
Running Test Method : test_case01
ok

----------------------------------------------------------------------
Ran 2 tests in 0.080s

OK
```

All the test classes are executed one by one in alphabetical order.

Test Fixtures

To put it simply, *test fixtures* are the set of steps performed before and after the tests. In `unittest`, these are implemented as methods of the `TestCase` class and can be overridden for your purposes. An example of custom test fixtures in `unittest` is shown in Listing 3-5,

Listing 3-5. test_module05.py

```
import unittest

def setUpModule():
        """called once, before anything else in this module"""
        print("In setUpModule()...")

def tearDownModule():
        """called once, after everything else in this module"""
        print("In tearDownModule()...")

class TestClass06(unittest.TestCase):

        @classmethod
        def setUpClass(cls):
                """called once, before any test"""
                print("In setUpClass()...")

        @classmethod
        def tearDownClass(cls):
                """called once, after all tests, if setUpClass successful"""
                print("In tearDownClass()...")

        def setUp(self):
                """called multiple times, before every test method"""
                print("\nIn setUp()...")

        def tearDown(self):
                """called multiple times, after every test method"""
                print("In tearDown()...")

        def test_case01(self):
                self.assertTrue("PYTHON".isupper())
                print("In test_case01()")

        def test_case02(self):
                self.assertFalse("python".isupper())
                print("In test_case02()")

if __name__ == '__main__':
        unittest.main()
```

In the code in Listing 3-5, the `setUpModule()` and `tearDownModule()`methods are the module-level fixtures. `setUpModule()` is executed before any method in the test module. `tearDownModule()` is executed after all methods in the test module. `setUpClass()` and `tearDownClass()` are class-level fixtures. `setUpClass()` is executed before any method in the test class. `tearDownClass()` is executed after all methods in the test class.

These methods are used with the `@classmethod` decorator, as shown in the code in Listing 3-5. The `@classmethod` decorator must have a reference to a class object as the first parameter. `setUp()` and `tearDown()` are method-level fixtures. `setUp()` and `tearDown()` methods are executed before and after every test method in the test class. Run the code in Listing 3-5 as follows:

```
python3 test_module05.py -v
```

The following is the output of the code:

```
In setUpModule()...
In setUpClass()...
test_case01 (__main__.TestClass06) ...
In setUp()...
In test_case01()
In tearDown()...
ok
test_case02 (__main__.TestClass06) ...
In setUp()...
In test_case02()
In tearDown()...
ok
In tearDownClass()...
In tearDownModule()...

----------------------------------------------------------------------
Ran 2 tests in 0.004s

OK
```

The test fixtures and their implementation is the key feature in any test automation library. This is a major advantage over the static testing offered by `doctest`.

Running Without unittest.main()

Up until now, you have run the test modules with `unittest.main()`. Now you will see how to run the test module without `unittest.main()`. Consider the code in Listing 3-6, for example.

Listing 3-6. test_module06.py

```
import unittest

class TestClass07(unittest.TestCase):

        def test_case01(self):
                self.assertTrue("PYTHON".isupper())
                print("\nIn test_case01()")
```

If you try to run it the usual way, with `python3 test_module06.py`, you do not get output in the console, as it does not have the `if __name__=='__main__'` and `unittest.main()` statements in it. Even running in verbose mode with `python3 test_module06.py -v` does not yield any output in the console.

The only way to run this module is to use the Python interpreter with the `-m unittest` option and the module name, as follows:

```
python -m unittest test_module06
```

The output is as follows:

```
In test_case01()
.
----------------------------------------------------------------------
Ran 1 test in 0.002s

OK
```

Note that you do not need to have `.py` after the module name as you did earlier. You can also enable verbosity with the `-v` options, as follows:

```
python -m unittest test_module06 -v
```

The verbose output is as follows:

```
test_case01 (test_module06.TestClass07) ...
In test_case01()
ok

----------------------------------------------------------------------
Ran 1 test in 0.002s

OK
```

We will use this same method throughout the chapter to run test modules. In later sections of this chapter, you will learn more about this method. For now, run all the previous code examples with this method of execution as an exercise.

Controlling the Granularity of Test Execution

You learned how to run a test module using the `-m` unittest option. You can also run individual test classes and test cases using this option.

Consider the earlier example of `test_module04.py` again, shown in Listing 3-7.

Listing 3-7. test_module04.py

```
import unittest
import inspect

class TestClass04(unittest.TestCase):
        def test_case01(self):
                print("\nClassname : " + self.__class__.__name__)
                print("Running Test Method : " + inspect.stack()[0][3])

class TestClass05(unittest.TestCase):

        def test_case01(self):
                print("\nClassname : " + self.__class__.__name__)
                print("Running Test Method : " + inspect.stack()[0][3])

if __name__ == '__main__':
        unittest.main(verbosity=2)
```

You can run the entire test module with the following command:

```
python3 -m unittest -v test_module04
```

The output is as follows:

```
test_case01 (test_module04.TestClass04) ...
Classname : TestClass04
Running Test Method : test_case01
ok
test_case01 (test_module04.TestClass05) ...
Classname : TestClass05
Running Test Method : test_case01
ok

----------------------------------------------------------------------
Ran 2 tests in 0.090s

OK
```

You can run a single test class with the following command:

```
python3 -m unittest -v test_module04.TestClass04
```

The output is as follows:

```
test_case01 (test_module04.TestClass04) ...
Classname : TestClass04
Running Test Method : test_case01
ok

----------------------------------------------------------------------
Ran 1 test in 0.077s

OK
```

You can also run a single test case with the following command:

```
python3 -m unittest -v test_module04.TestClass05.test_case01
```

The output is as follows:

```
test_case01 (test_module04.TestClass05) ...
Classname : TestClass05
Running Test Method : test_case01
ok

----------------------------------------------------------------------
Ran 1 test in 0.077s

OK
```

This way you can control the granularity of the test execution.

Listing All the Command-Line Options and Help

You can list all the command line options of `unittest` using the `-h` command-line option. Run the following command:

```
python3 -m unittest -h
```

The following is the output:

```
usage: python3 -m unittest [-h] [-v] [-q] [-f] [-c] [-b] [tests [tests ...]]

positional arguments:
  tests           a list of any number of test modules, classes and test
                  methods.

optional arguments:
  -h, --help      show this help message and exit
  -v, --verbose   Verbose output
  -q, --quiet     Quiet output
  -f, --failfast  Stop on first fail or error
  -c, --catch     Catch ctrl-C and display results so far
  -b, --buffer    Buffer stdout and stderr during tests

Examples:
  python3 -m unittest test_module               - run tests from test_module
  python3 -m unittest module.TestClass          - run tests from module.
                                                  TestClass
  python3 -m unittest module.Class.test_method  - run specified test method
```

```
usage: python3 -m unittest discover [-h] [-v] [-q] [-f] [-c] [-b] [-s START]
                                    [-p PATTERN] [-t TOP]

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         Verbose output
  -q, --quiet           Quiet output
  -f, --failfast        Stop on first fail or error
  -c, --catch           Catch ctrl-C and display results so far
  -b, --buffer          Buffer stdout and stderr during tests
  -s START, --start-directory START
                        Directory to start discovery ('.' default)
  -p PATTERN, --pattern PATTERN
                        Pattern to match tests ('test*.py' default)
  -t TOP, --top-level-directory TOP
                        Top level directory of project (defaults to start
                        directory)

For test discovery all test modules must be importable from the top level
directory of the project.
```

This way you get a detailed summary of the various command-line options available with unittest.

Important Command-Line Options

Let's take a look at the important command-line options in unittest. Take a look at the code in Listing 3-8 for example.

Listing 3-8. test_module07.py

```
import unittest

class TestClass08(unittest.TestCase):

        def test_case01(self):
                self.assertTrue("PYTHON".isupper())
                print("\nIn test_case1()")

        def test_case02(self):
                self.assertTrue("Python".isupper())
                print("\nIn test_case2()")

        def test_case03(self):
                self.assertTrue(True)
                print("\nIn test_case3()")
```

You already know that -v stands for verbose mode. The following is the output in verbose mode:

```
test_case01 (test_module07.TestClass08) ...
In test_case1()
ok
test_case02 (test_module07.TestClass08) ... FAIL
test_case03 (test_module07.TestClass08) ...
In test_case3()
ok

======================================================================
FAIL: test_case02 (test_module07.TestClass08)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/pi/book/code/chapter03/test/test_module07.py", line 11, in
test_case02
    self.assertTrue("Python".isupper())
AssertionError: False is not true

----------------------------------------------------------------------
Ran 3 tests in 0.012s

FAILED (failures=1)
```

The option -q stands for *quiet* mode. Run the following command to demonstrate quiet mode:

```
python3 -m unittest -q test_module07
```

The output is as follows:

```
In test_case1()

In test_case3()
======================================================================
FAIL: test_case02 (test_module07.TestClass08)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/pi/book/code/chapter03/test/test_module07.py", line 11, in
  test_case02
    self.assertTrue("Python".isupper())
AssertionError: False is not true

----------------------------------------------------------------------
Ran 3 tests in 0.005s

FAILED (failures=1)
```

The option -f stands for *failsafe*. It forcefully stops execution as soon as the first test case fails. Run the following command to initiate failsafe mode:

```
python3 -m unittest -q test_module07
```

The following is the output in failsafe mode:

```
In test_case1()
.F
======================================================================
FAIL: test_case02 (test_module07.TestClass08)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/pi/book/code/chapter03/test/test_module07.py", line 11, in
  test_case02
    self.assertTrue("Python".isupper())
AssertionError: False is not true

----------------------------------------------------------------------
Ran 2 tests in 0.004s

FAILED (failures=1)
```

You can also use more than one option. For example, you can combine verbose with failsafe using the following command:

```
python3 -m unittest -fv test_module07
```

The output is as follows:

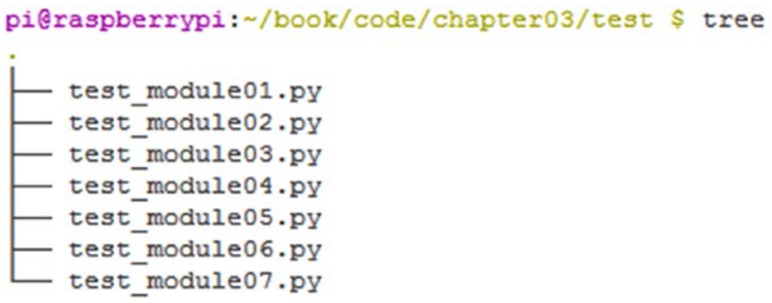
```
test_case01 (test_module07.TestClass08) ...
In test_case1()
ok
test_case02 (test_module07.TestClass08) ... FAIL

======================================================================
FAIL: test_case02 (test_module07.TestClass08)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/pi/book/code/chapter03/test/test_module07.py", line 11, in
test_case02
    self.assertTrue("Python".isupper())
AssertionError: False is not true

----------------------------------------------------------------------
Ran 2 tests in 0.005s

FAILED (failures=1)
```

As an exercise, try to use different combinations of command-line options.

Creating a Test Package

Up until now, you have created and executed test modules individually. However, you can use Python's built-in packaging feature to create a package of tests. This is standard practice in complex projects with large codebases.

Figure 3-1 shows a snapshot of the current `test` directory where you are saving your test modules.



```
pi@raspberrypi:~/book/code/chapter03/test $ tree
.
├── test_module01.py
├── test_module02.py
├── test_module03.py
├── test_module04.py
├── test_module05.py
├── test_module06.py
└── test_module07.py
```

Figure 3-1. *Snapshot of the test subdirectory in the chapter03 directory*

Now, let's create a package of test modules. Create an `__init__.py` file in the `test` directory. Add the code in Listing 3-9 to the `__init__.py` file,

Listing 3-9. __init__.py

```
all = ["test_module01", "test_module02", "test_module03", "test_module04",
"test_module05", "test_module06", "test_module07"]
```

Congratulations! You just created a test package. `test` is the name of the testing package and all modules mentioned in the `__init__.py` belong to this package. If you need to add a new testing module to the package `test`, you need to create a new test module file in the test directory and then add the name of that module to the `__init__.py` file.

Now you can run the test modules from the parent directory of `test` (`chapter03`) in the following way. Move to the `chapter03` directory using the following command:

```
cd /home/pi/book/code/chapter03
```

Note that the path might be different in your case, depending on where you have created the book directory.

Run the `test` module with the following command:

```
python3 -m unittest -v test.test_module04
```

The following is the output:

```
test_case01 (test.test_module04.TestClass04) ...
Classname : TestClass04
Running Test Method : test_case01
ok
test_case01 (test.test_module04.TestClass05) ...
Classname : TestClass05
Running Test Method : test_case01
ok

----------------------------------------------------------------------
Ran 2 tests in 0.090s

OK
```

Run a test class in the test module with the following command:

```
python3 -m unittest -v test.test_module04.TestClass04
```

The output is as follows:

```
test_case01 (test.test_module04.TestClass04) ...
Classname : TestClass04
Running Test Method : test_case01
ok

----------------------------------------------------------------------
Ran 1 test in 0.078s

OK
```

Run a test case from a test module as follows:

```
python3 -m unittest -v
test.test_module04.TestClass04.test_case01
```

The output is as follows:

```
test_case01 (test.test_module04.TestClass04) ...
Classname : TestClass04
Running Test Method : test_case01
ok

----------------------------------------------------------------------
Ran 1 test in 0.079s

OK
```

Organizing the Code

Let's look at the ways you can organize the test code and dev. We're now moving toward real-life project scenarios for using `unittest`. Up until now, the tests (the testing code) and the code to be tested (the development code) were in the same module. Usually in the real-life projects, the development code and the test code are kept in different files.

Placing the Development and Test Code in a Single Directory

Here, you will organize the dev and test code in a single directory. In the `test` directory, create a module called `test_me.py` and add the code in Listing 3-10 to it.

Listing 3-10. test_me.py

```
def add(x, y):
    return(x + y)

def mul(x, y):
    return(x * y)

def sub(x, y):
    return(x - y)

def div(x, y):
    return(x / y)
```

Now, since `test_me.py` is in the `test` directory, it can directly be imported into another module in the same directory using the `import test_me` statement. The test module in Listing 3-11 imports `test_me.py` to test its functionality.

Listing 3-11. test_module08.py

```
import unittest
import test_me

class TestClass09(unittest.TestCase):

        def test_case01(self):
                self.assertEqual(test_me.add(2, 3), 5)
                print("\nIn test_case01()")

        def test_case02(self):
                self.assertEqual(test_me.mul(2, 3), 6)
                print("\nIn test_case02()")
```

Run the test module with the following command:

```
python3 -m unittest -v test_module08
```

The output is as follows:

```
test_case01 (test_module08.TestClass09) ...
In test_case01()
ok
test_case02 (test_module08.TestClass09) ...
In test_case02()
ok

----------------------------------------------------------------------
Ran 2 tests in 0.004s

OK
```

This way, you can organize the development code and the testing code in the same directory, in the different files.

Placing the Development and Test Code in Separate Directories

Many coding standards recommend that the development code and the testing code files be organized in separate directories. Let's do that now.

Navigate to the `chapter03` directory,

```
cd /home/pi/book/code/chapter03
```

Create a new directory called `mypackage` in the `chapter03` directory:

```
mkdir mypackage
```

Navigate to the `mypackage` directory:

```
cd mypackage
```

Save the code in Listing 3-12 as the file `mymathlib.py` in the `mypackage` directory,

Listing 3-12. mymathlib.py

```
class mymathlib:
        def __init__(self):
                """Constructor for this class..."""
                print("Creating object : " + self.__class__.__name__)
        def add(self, x, y):
                return(x + y)

        def mul(self, x, y):
                return(x * y)

        def mul(self, x, y):
                return(x - y)

        def __del__(self):
                """Destructor for this class..."""
                print("Destroying object : " + self.__class__.__name__)
```

Save the code in Listing 3-13 as the file `mymathsimple.py` in the `mypackage` directory.

Listing 3-13. mymathsimple.py

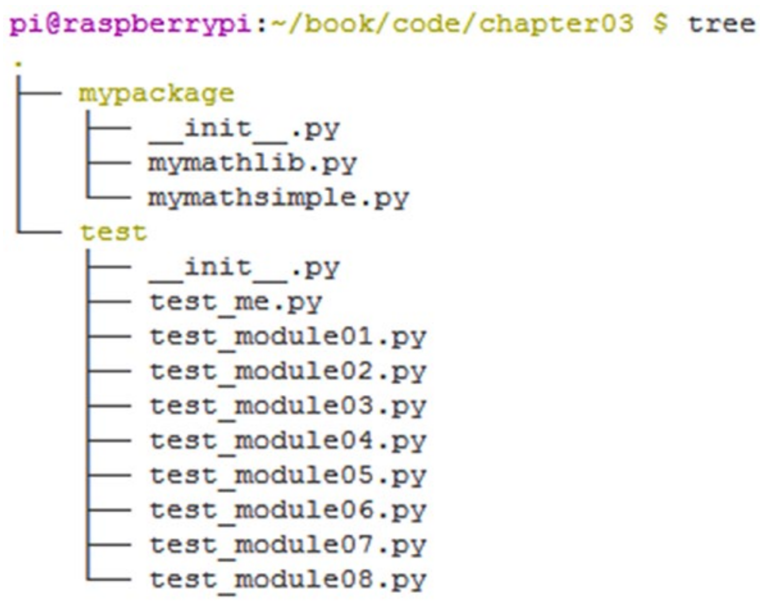
```
def add(x, y):
    return(x + y)

def mul(x, y):
    return(x * y)

def sub(x, y):
    return(x - y)

def div(x, y):
    return(x / y)
```

These modules you just created are the development modules. Finally, to create a package of the development modules, create the `__init__.py` file with the code shown in Listing 3-14.

Listing 3-14. __init__.py

```
all = ["mymathlib", "mymathsimple"]
```

This will create a Python package for the development code. Now, navigate back to the `chapter03` directory. The structure of the `chapter03` directory should now look like Figure 3-2.



```
pi@raspberrypi:~/book/code/chapter03 $ tree
.
├── mypackage
│   ├── __init__.py
│   ├── mymathlib.py
│   └── mymathsimple.py
└── test
    ├── __init__.py
    ├── test_me.py
    ├── test_module01.py
    ├── test_module02.py
    ├── test_module03.py
    ├── test_module04.py
    ├── test_module05.py
    ├── test_module06.py
    ├── test_module07.py
    └── test_module08.py
```

Figure 3-2. *Snapshot of the chapter03 directory*

`mypackage` is the package of the development code and `test` is the package of the testing code.

You now need to create a test module for testing the development code in `mypackage`. Create a new test module called `test_module09.py` in the `test` directory and add the code shown in Listing 3-15.

Listing 3-15. test_module09.py

```
from mypackage.mymathlib import *
import unittest

math_obj = 0

def setUpModule():
        """called once, before anything else in the module"""
        print("In setUpModule()...")
        global math_obj
        math_obj = mymathlib()

def tearDownModule():
        """called once, after everything else in the module"""
        print("In tearDownModule()...")
        global math_obj
        del math_obj
```

```
class TestClass10(unittest.TestCase):

        @classmethod
        def setUpClass(cls):
                """called only once, before any test in the class"""
                print("In setUpClass()...")

        def setUp(self):
                """called once before every test method"""
                print("\nIn setUp()...")

        def test_case01(self):
                print("In test_case01()")
                self.assertEqual(math_obj.add(2, 5), 7)

        def test_case02(self):
                print("In test_case02()")

        def tearDown(self):
                """called once after every test method"""
                print("In tearDown()...")

        @classmethod
        def tearDownClass(cls):
                """called once, after all the tests in the class"""
                print("In tearDownClass()...")
```

Add `test_module09` to `__init__.py` in the `test` directory to make it part of the `test` package.

Run the code from the `test` directory using the following command:

```
python3 -m unittest -v test_module09
```

It will throw an error as follows:

```
from mypackage.mymathlib import *
ImportError: No module named 'mypackage'
```

That's because the `mypackage` module is not visible from the `test` directory. It lives not in the `test` directory, but in the `chapter03` directory. This module cannot be executed from the `test` directory. You must execute this module as a part of the `test` package. You can do this from the `chapter03` directory. The `mypackage` module is visible in this directory as `mypackage`, which is a subdirectory of `chapter03`.

Navigate to the `chapter03` directory and run this module as follows:

```
python3 -m unittest -v test.test_module09
```

Here is the output of the execution:

```
In setUpModule()...
Creating object : mymathlib
In setUpClass()...
test_case01 (test.test_module09.TestClass10) ...
In setUp()...
In test_case01()
In tearDown()...
ok
test_case02 (test.test_module09.TestClass10) ...
In setUp()...
In test_case02()
In tearDown()...
ok
In tearDownClass()...
In tearDownModule()...
Destroying object : mymathlib

----------------------------------------------------------------------
Ran 2 tests in 0.004s

OK
```

That's how you organize the development and testing code files in separate directories. It is standard practice to separate these code files.

Test Discovery

Test discovery is the process of discovering and executing all the tests in the project directory and all its subdirectories. The test discovery process is automated in `unittest` and can be invoked using the `discover` sub-command. It can be invoked with the following command:

```
python3 -m unittest discover
```

Here is the partial output of this command when it runs in the `chapter02` directory:

```
..
Running Test Method : test_case01
.
Running Test Method : test_case02
.
Running Test Method : test_case01
We're in custom made function : add
.
```

```
Running Test Method : test_case02
.
Running Test Method : test_case03
We're in custom made function : add
F
Running Test Method : test_case04
F
Classname : TestClass04
Running Test Method : test_case01
```

You can also invoke it using the verbose mode with the following command:

```
python3 -m unittest discover -v
```

Here is the partial output of this command:

```
test_case01 (test.test_module01.TestClass01) ... ok
test_case02 (test.test_module01.TestClass01) ... ok
test_case01 (test.test_module02.TestClass02) ...
Running Test Method : test_case01
ok
test_case02 (test.test_module02.TestClass02) ...
Running Test Method : test_case02
ok
test_case01 (test.test_module03.TestClass03) ...
Running Test Method : test_case01
We're in custom made function : add
ok
test_case02 (test.test_module03.TestClass03) ...
Running Test Method : test_case02
ok
test_case03 (test.test_module03.TestClass03) ...
Running Test Method : test_case03
We're in custom made function : add
```

As you can see in the verbose output, the `unittest` automatically found and ran all the test modules located in the `chapter03` directory and its subdirectories. This saves you the pain of running each test module separately and collecting the results individually. Test discovery is one of the most important features of any automation testing framework.

Coding Conventions for unittest

As you have seen, test discovery automatically finds and runs all the tests in a project directory. To achieve this effect, you need to follow some coding and naming conventions for your test code. You may have noticed already that I have consistently followed these conventions in all the code examples in this book.

- In order to be compatible with test discovery, all of the test files must be either modules or packages importable from the top-level directory of the project.
- By default, the test discovery always starts from the current directory.
- By default, test discovery always searches for `test*.py` patterns in the filenames.

Assertions in unittest

You have learned about a few basic assertions, like `assertEqual()` and `assertTrue()`. The following tables list the most used assertions and their purpose.

Method	Checks That
`assertEqual(a, b)`	`a == b`
`assertNotEqual(a, b)`	`a != b`
`assertTrue(x)`	`bool(x) is True`
`assertFalse(x)`	`bool(x) is False`
`assertIs(a, b)`	`a is b`
`assertIsNot(a, b)`	`a is not b`
`assertIsNone(x)`	`x is None`
`assertIsNotNone(x)`	`x is not None`
`assertIn(a, b)`	`a in b`
`assertNotIn(a, b)`	`a not in b`
`assertIsInstance(a, b)`	`isinstance(a, b)`
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`

Method	Checks That
`assertAlmostEqual(a, b)`	`round(a-b, 7) == 0`
`assertNotAlmostEqual(a, b)`	`round(a-b, 7) != 0`
`assertGreater(a, b)`	`a > b`
`assertGreaterEqual(a, b)`	`a >= b`
`assertLess(a, b)`	`a < b`
`assertLessEqual(a, b)`	`a <= b`
`assertRegexpMatches(s, r)`	`r.search(s)`
`assertNotRegexpMatches(s, r)`	`not r.search(s)`
`assertItemsEqual(a, b)`	`sorted(a) == sorted(b)`
`assertDictContainsSubset(a, b)`	`all the key/value pairs in a exist in b`

Method	Used to Compare
`assertMultiLineEqual(a, b)`	`strings`
`assertSequenceEqual(a, b)`	`sequences`
`assertListEqual(a, b)`	`lists`
`assertTupleEqual(a, b)`	`tuples`
`assertSetEqual(a, b)`	`sets or frozensets`
`assertDictEqual(a, b)`	`dicts`

All the `assert` methods listed in the previous tables are good enough for most of the programmers and testers for automating the tests.

Other Useful Methods

This section looks at a few useful methods that will help you debug and understand the flow of execution.

The `id()` and `shortDescription()` methods are very useful for debugging. `id()` returns the name of the method and `shortDescription()` returns the description of the method. Listing 3-16 shows an example.

Listing 3-16. test_module10.py

```
import unittest

class TestClass11(unittest.TestCase):

        def test_case01(self):
                """This is a test method..."""
                print("\nIn test_case01()")
                print(self.id())
                print(self.shortDescription())
```

The output of Listing 3-16 is as follows:

```
test_case01 (test_module10.TestClass11)
This is a test method... ...
In test_case01()
test_module10.TestClass11.test_case01
This is a test method...
ok

----------------------------------------------------------------------
Ran 1 test in 0.002s

OK
```

Failing a Test

Many times, you might want to have a method that explicitly fails a test when it's called. In `unittest`, the `fail()` method is used for that purpose. Check the code in Listing 3-17 as an example.

Listing 3-17. test_module11.py

```
import unittest

class TestClass12(unittest.TestCase):

        def test_case01(self):
                """This is a test method..."""
                print(self.id())
                self.fail()
```

The output of Listing 3-16 is as follows:

```
test_case01 (test_module11.TestClass12)
This is a test method... ...
test_module11.TestClass12.test_case01
FAIL

======================================================================
FAIL: test_case01 (test_module11.TestClass12)
This is a test method...
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/pi/book/code/chapter03/test/test_module11.py", line 9, in
test_case01
    self.fail()
AssertionError: None

----------------------------------------------------------------------
Ran 1 test in 0.004s

FAILED (failures=1)

Skipping tests
```

`unittest` provides a mechanism for skipping tests, conditionally or unconditionally. It uses the following decorators for implementing the skipping mechanism:

- `unittest.skip(reason)`: Unconditionally skips the decorated test. `reason` should describe why the test is being skipped.
- `unittest.skipIf(condition, reason)`: Skips the decorated test if `condition` is `true`.

- `unittest.skipUnless(condition, reason)`: Skips the decorated test unless `condition` is `true`.
- `unittest.expectedFailure()`: Marks the test as an expected failure. If the test fails when it runs, the test is not counted as a failure.

The code in Listing 3-18 demonstrates how to skip tests conditionally and unconditionally.

Listing 3-18. test_module12.py

```
import sys
import unittest

class TestClass13(unittest.TestCase):

    @unittest.skip("demonstrating unconditional skipping")
    def test_case01(self):
        self.fail("FATAL")

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
    def test_case02(self):
        # Windows specific testing code
        pass

    @unittest.skipUnless(sys.platform.startswith("linux"), "requires Linux")
    def test_case03(self):
        # Linux specific testing code
        pass
```

When you run the code in Listing 3-18 on the Linux platform, the output is as follows:

```
test_case01 (test_module12.TestClass13) ... skipped 'demonstrating
unconditional skipping'
test_case02 (test_module12.TestClass13) ... skipped 'requires Windows'
test_case03 (test_module12.TestClass13) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.003s

OK (skipped=2)
```

When you run the code in Listing 3-18 on the Windows platform, the output is as follows:

```
test_case01 (test_module12.TestClass13) ... skipped 'demonstrating
unconditional skipping'
```

```
test_case02 (test_module12.TestClass13) ... ok
test_case03 (test_module12.TestClass13) ... skipped
'requires Linux'

----------------------------------------------------------------------
Ran 3 tests in 0.003s

OK (skipped=2)
```

As you can see, the code skips the test cases based on the OS where it runs. This trick is very useful for running platform-specific test cases.

You can also skip entire test classes in a test module using the `unittest.skip(reason)` decorator.

Exceptions in the Test Case

When an exception is raised in a test case, the test case fails. The code shown in Listing 3-19 will raise an exception explicitly.

Listing 3-19. test_module13.py

```
import unittest

class TestClass14(unittest.TestCase):
    def test_case01(self):
        raise Exception
```

The output of Listing 3-19 is as follows:

```
test_case01 (test_module13.TestClass14) ... ERROR

======================================================================
ERROR: test_case01 (test_module13.TestClass14)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/pi/book/code/chapter03/test/test_module13.py", line 6, in
  test_case01
    raise Exception
Exception

----------------------------------------------------------------------
Ran 1 test in 0.004s

FAILED (errors=1)
```

The failure message shown when the test fails due to an exception is different from when the test fails due to an assertion.

assertRaises()

You learned that the assert methods are used to check the test conditions. The assertRaises() method is used to check if the code block raises the exception mentioned in assertRaises(). If the code raises the exception then the test passes; otherwise, it fails. The code shown in Listing 3-20 demonstrates the usage of assertRaises() in detail.

Listing 3-20. test_module14.py

```
import unittest

class Calculator:

        def add1(self, x, y):
                return x + y

        def add2(self, x, y):
                number_types = (int, float, complex)
                if isinstance(x, number_types) and isinstance(y, number_
                types):
                        return x + y
                else:
                        raise ValueError

calc = 0

class TestClass16(unittest.TestCase):

        @classmethod
        def setUpClass(cls):
                global calc
                calc = Calculator()

        def setUp(self):
                print("\nIn setUp()...")

        def test_case01(self):
                self.assertEqual(calc.add1(2, 2), 4)

        def test_case02(self):
                self.assertEqual(calc.add2(2, 2), 4)

        def test_case03(self):
                self.assertRaises(ValueError, calc.add1, 2, 'two')

        def test_case04(self):
                self.assertRaises(ValueError, calc.add2, 2, 'two')
        def tearDown(self):
                print("\nIn tearDown()...")

        @classmethod
        def tearDownClass(cls):
                global calc
                del calc
```

In the code in Listing 3-20, we defined a class called `Calculator` that has two different methods for the addition operation. The `add1()` method does not have a provision to raise an exception if a non-numeric argument is passed to it. The `add2()` method raises a `ValueError` if any of the arguments are non-numeric. Here is the output of the code in Listing 3-20:

```
test_case01 (test_module14.TestClass16) ...
In setUp()...

In tearDown()...
ok
test_case02 (test_module14.TestClass16) ...
In setUp()...

In tearDown()...
ok
test_case03 (test_module14.TestClass16) ...
In setUp()...

In tearDown()...
ERROR
test_case04 (test_module14.TestClass16) ...
In setUp()...

In tearDown()...
ok

======================================================================
ERROR: test_case03 (test_module14.TestClass16)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/pi/book/code/chapter03/test/test_module14.py", line 37, in
  test_case03
    self.assertRaises(ValueError, calc.add1, 2, 'two')
  File "/usr/lib/python3.4/unittest/case.py", line 704, in assertRaises
    return context.handle('assertRaises', callableObj, args, kwargs)
  File "/usr/lib/python3.4/unittest/case.py", line 162, in handle
    callable_obj(*args, **kwargs)
  File "/home/pi/book/code/chapter03/test/test_module14.py", line 7, in add1
    return x + y
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'

----------------------------------------------------------------------
Ran 4 tests in 0.030s

FAILED (errors=1)
```

In the output, the `test_Case03()` fails because `add1()` does not have a provision to raise an exception when you pass it a non-numeric argument (a string, in this case). `assertRaises()` is very useful in writing negative test cases, such as when you need to check the behavior of the API against invalid arguments.

EXERCISE 3-1

`unittest`, like all the other Python libraries, is too vast a topic to be covered in a single book. So, I recommend you complete the following exercises to gain more knowledge and experience with `unittest`.

1. Visit the Python 3 Documentation page for `unittest` at `https://docs.python.org/3/library/unittest.html`.
2. Practice all the assertion methods mentioned in this chapter by writing tests using each one of them.
3. Practice using the `unittest.skipIf(condition, reason)` and `unittest.expectedFailure()` decorators. Write code to demonstrate their functionality.
4. Write a test module with multiple test classes and skip an entire test class using the `unittest.skip(reason)` decorator.
5. Experiment with raising exceptions in the test fixtures.

   Hint: Try to run the code in Listing 3-21 by enabling each commented-out `raise Exception` line, one line at a time. This will help you understand how an individual fixture behaves when you raise an exception in it.

Listing 3-21. `test_module15.py`

```
import unittest

def setUpModule():
#   raise Exception
    pass
```

```
def tearDownModule():
#   raise Exception
    pass

class TestClass15(unittest.TestCase):

    @classmethod
    def setUpClass(cls):
#       raise Exception
        pass

    def setUp(self):
#       raise Exception
        pass

    def test_case01(self):
        self.id()

    def tearDown(self):
#       raise Exception
        pass

    @classmethod
    def tearDownClass(cls):
#       raise Exception
        pass
```

Conclusion

In this chapter, you learned about several important concepts, including test fixtures, test classes, test methods, and test modules. You also learned how to implement all these concepts with `unittest`. Almost all the concepts you learned in this chapter will be revisited in later chapters that cover other Python testing frameworks. In the next chapter, we will look at `nose` and `nose2`, which are two other popular Python test automation frameworks.

CHAPTER 4

nose and nose2

The last chapter introduced `xUnit` and `unittest`. In this chapter, we will explore yet another unit testing API for Python, called `nose`. The tagline of `nose` is, *nose extends unittest to make testing easier.*

You can use `nose`'s API to write and run automated tests. You can also use `nose` to run tests written in other frameworks like `unittest`. This chapter will also explore the next actively developed and maintained iteration of `nose`, `nose2`.

Introduction to nose

`nose` is not the part of Python's standard library. You have to install it in order to use it. Let's see how we can install it on Python 3.

Installing nose on Linux OS

The easiest way to install `nose` on a Linux computer is to install it using Python's package manager `pip.` `Pip` stands for *pip installs packages*. It's a recursive acronym. If `pip` is not installed on your Linux computer, you can install it by using a system package manager. On any Debian/Ubuntu or derivative computer, install `pip` with the following command:

```
sudo apt-get install python3-pip
```

On Fedora/CentOS and derivatives, run following commands (assuming you have Python 3.5 installed on the OS) to install `pip`:

```
sudo yum install python35-setuptools
sudo easy_install pip
```

Once `pip` is installed, you can install `nose` with the following command:

```
sudo pip3 install nose
```

Installing nose on MacOS and Windows

`pip` is pre-installed with Python 3 on MacOS and Windows. Install `nose` with the following command:

```
pip3 install nose
```

Verifying the Installation

Once `nose` is installed, run the following command to verify the installation:

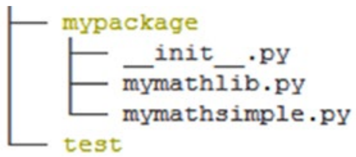
```
nosetests -V
```

It will show output as follows:

```
nosetests version 1.3.7
```

Getting Started with nose

To get started with `nose`, follow the same path of exploration that you followed with `unittest`. Create a directory called `chapter04` in the `code` directory and copy the `mypackage` directory from the `chapter03` directory to `code`. You will need it later. Create a directory called `test` too. After all this, the `chapter04` directory structure should look like the structure shown in Figure 4-1.



```
├── mypackage
│   ├── __init__.py
│   ├── mymathlib.py
│   └── mymathsimple.py
└── test
```

Figure 4-1. *The chapter04 directory structure*

Save all the code examples to the `test` directory only.

A Simple nose Test Case

A very simple `nose` test case is demonstrated in Listing 4-1.

Listing 4-1. test_module01.py

```
def test_case01():
        assert 'aaa'.upper() == 'AAA'
```

In Listing 4-1, `test_case01()` is the test function. `assert` is Python's built-in keyword and it works like the `assert` methods in `unittest`. If you compare this code with the simplest test case in the `unittest` framework, you will notice that you do not have to extend the test from any parent class. This makes the test code cleaner and less cluttered.

If you try to run it with the following commands, it will not yield any output:

```
python3 test_module01.py
python3 test_module01.py -v
```

This is because you have not included a test-runner in the code.

You can run it by using the `-m` command-line option for Python as follows:

```
python3 -m nose test_module01.py
```

The output is as follows:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.007s

OK
```

Verbose mode can be invoked by adding the `-v` command-line option as follows:

```
python3 -m nose -v test_module01.py
```

The output is as follows:

```
test.test_module01.test_case01 ... ok
----------------------------------------------------------------------
Ran 1 test in 0.007s

OK
```

Running the Test Module with nosetests

You can use nose's `nosetests` command to run the test modules as follows:

```
nosetests test_module01.py
```

The output is as follows:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.006s

OK
```

Verbose mode can be invoked as follows:

```
nosetests test_module01.py -v
```

The output is as follows:

```
test.test_module01.test_case01 ... ok
----------------------------------------------------------------------
Ran 1 test in 0.007s

OK
```

Using the `nosetests` command is the simplest way to run the test modules. Due to the simplicity and convenience of the coding and invocation style, we will use `nosetests` to run the tests until we introduce and explain `nose2`.

Getting Help

Use the following command to get help and documentation about `nose`:

```
nosetests -h
```

Organizing the test code

In the previous chapter, you learned how to organize the development and the testing code of the project in separate directories. You will follow the same standard in this and the next chapter too. First create a test module to test the development code in `mypackage`. Save the code shown in Listing 4-2 in the `test` directory.

Listing 4-2. test_module02.py

```
from mypackage.mymathlib import *

class TestClass01:
        def test_case01(self):
                print("In test_case01()")
                assert mymathlib().add(2, 5) == 7
```

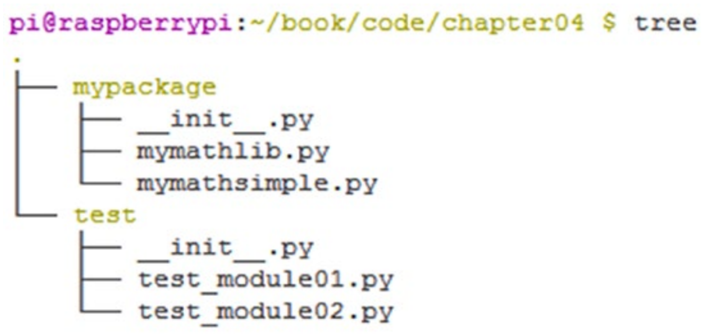
Listing 4-2 creates a test class called `TestClass01`. As discussed earlier, you do not have to extend it from a parent class. The line containing `assert` checks if the statement `mymathlib(). add(2, 5) == 7` is `true` or `false` to mark the test method as `PASS` or `FAIL`.

Also, create an `__init__.py` file with the code in Listing 4-3 placed in the `test` directory.

Listing 4-3. __init__.py

```
all = ["test_module01", "test_module02"]
```

After this, the chapter04 directory structure will resemble Figure 4-2.



```
pi@raspberrypi:~/book/code/chapter04 $ tree
.
├── mypackage
│   ├── __init__.py
│   ├── mymathlib.py
│   └── mymathsimple.py
└── test
    ├── __init__.py
    ├── test_module01.py
    └── test_module02.py
```

Figure 4-2. *The chapter04 directory structure*

The test package is ready now. You can run the tests from the chapter04 directory as follows:

```
nosetests test.test_module02 -v
```

The output is as follows:

```
test.test_module02.TestClass01.test_case01 ... ok
----------------------------------------------------------------------
Ran 1 test in 0.008s
OK
```

The convention for running a specific test class is a bit different in nose. The following is the example:

```
nosetests test.test_module02:TestClass01 -v
```

You can also run an individual test method as follows:

```
nosetests test.test_module02:TestClass01.test_case01 -v
```

Test Discovery

You learned about test discovery in an earlier chapter. nose also supports the test discovery process. In fact, test discovery in nose is even simpler than in unittest. You do not have to use the discover sub-command for test discovery. You just need to navigate to the project directory (chapter04 in this case) and run the nosetests command, as follows:

```
nosetests
```

You can also invoke this process in verbose mode:

```
nosetests -v
```

The output is as follows:

```
test.test_module01.test_case01 ... ok
test.test_module02.TestClass01.test_case01 ... ok
Ran 2 tests in 0.328s
OK
```

As you can see in the output, `nosetests` automatically discovers the test package and runs all its test modules.

Fixtures for Classes, Modules, and Methods

`nose` provides `xUnit`-style fixtures that behave in similar way as the fixtures in `unittest`. Even the names of the fixtures are same. Consider the code in Listing 4-4.

Listing 4-4. test_module03.py

```
from mypackage.mymathlib import *

math_obj = 0

def setUpModule():
    """called once, before anything else in this module"""
    print("In setUpModule()...")
    global math_obj
    math_obj = mymathlib()

def tearDownModule():
    """called once, after everything else in this module"""
    print("In tearDownModule()...")
    global math_obj
    del math_obj

class TestClass02:

    @classmethod
    def setUpClass(cls):
                """called once, before any test in the class"""
                print("In setUpClass()...")

            def setUp(self):
                """called before every test method"""
                print("\nIn setUp()...")

            def test_case01(self):
                print("In test_case01()")
                assert math_obj.add(2, 5) == 7

            def test_case02(self):
                print("In test_case02()")

            def tearDown(self):
                """called after every test method"""
                print("In tearDown()...")

            @classmethod
            def tearDownClass(cls):
                """called once, after all tests, if setUpClass() successful"""
                print ("\nIn tearDownClass()...")
```

If you run the code in Listing 4-4 with the following command:

```
nosetests test_module03.py -v
```

The output will be as follows:

```
test.test_module03.TestClass02.test_case01 ... ok
test.test_module03.TestClass02.test_case02 ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.010s

OK
```

In order to get more details of test execution, you need to add the `-s` option to the command line, which allows any `stdout` output to be printed in the command line immediately.

Run the following command:

```
nosetests test_module03.py -vs
```

The output is as follows:

```
In setUpModule()...
Creating object : mymathlib
In setUpClass()...
test.test_module03.TestClass02.test_case01 ...
In setUp()...
In test_case01()
In tearDown()...
ok
test.test_module03.TestClass02.test_case02 ...
In setUp()...
In test_case02()
In tearDown()...
ok

In tearDownClass()...
In tearDownModule()...
Destroying object : mymathlib

----------------------------------------------------------------------
Ran 2 tests in 0.011s

OK
```

From now onward, we will add the `-s` option to the `nosetests` command while executing the tests.

Fixtures for Functions

Before you get started with the fixtures for functions, you must understand the difference between a function and a method in Python. A *function* is a named piece of code that performs an operation and a *method* is a function with an extra parameter that's the object on which it runs. A function is not associated with a class. A method is always associated with a class.

Check the code in Listing 4-5 as an example.

Listing 4-5. test_module04.py

```
from nose.tools import with_setup

def setUpModule():
    """called once, before anything else in this module"""
    print("\nIn setUpModule()...")
```

```
def tearDownModule():
    """called once, after everything else in this module"""
    print("\nIn tearDownModule()...")

def setup_function():
    """setup_function(): use it with @with_setup() decorator"""
    print("\nsetup_function()...")

def teardown_function():
    """teardown_function(): use it with @with_setup() decorator"""
    print("\nteardown_function()...")

def test_case01():
    print("In test_case01()...")

def test_case02():
    print("In test_case02()...")

@with_setup(setup_function, teardown_function)
def test_case03():
    print("In test_case03()...")
```

In the code in Listing 4-5, `test_case01()`, `test_case02()`, `test_case03()`, `setup_function()`, and `teardown_function()` are the functions. They are not associated with a class. You have to use the `@with_setup()` decorator, which is imported from `nose.tools`, for assigning `setup_function()` and `teardown_function()` as fixtures of `test_case03()`. nose recognizes `test_case01()`, `test_case02()`, and `test_case03()` as test functions because the names begin with `test_`. `setup_function()` and `teardown_function()` are recognized as fixtures of `test_case03()`, due to the `@with_setup()` decorator.

The `test_case01()` and `test_case02()`functions do not have any fixtures assigned to them.

Let's run this code with the following command:

```
nosetests test_module04.py -vs
```

The output is as follows:

```
In setUpModule()...
test.test_module04.test_case01 ... In test_case01()...
ok
test.test_module04.test_case02 ... In test_case02()...
ok
test.test_module04.test_case03 ...
setup_function()...
In test_case03()...

teardown_function()...
ok
```

```
In tearDownModule()...

----------------------------------------------------------------------
Ran 3 tests in 0.011s

OK
```

As you can see in the output, `setup_function()` and `teardown_function()` run before and after `test_case03()`, respectively. `unittest` does not have any provision for the fixtures at the test function level. Actually, `unittest` does not support the concept of standalone test functions, as everything has to be extended from the `TestCase` class and a function cannot be extended.

It's not mandatory that you name the function-level fixtures `setup_function()` and `teardown_function()`. You can name that anything you want (except, of course, for Python 3's reserved keywords). Those will be executed before and after the test function as long as you use those in the `@with_setup()` decorator.

Fixtures for Packages

`unittest` does not have a provision for package-level fixtures. Package fixtures are executed when the test package or part of the test package is invoked. Change the contents of the `__init__.py` file in the `test` directory to the code in Listing 4-6.

Listing 4-6. __init__.py

```
all = ["test_module01", "test_module02", "test_module03", "test_module04"]

def setUpPackage():
    print("In setUpPackage()...")

def tearDownPackage():
    print("In tearDownPackage()...")
```

If you run a module in this package now, the package-level fixtures will run before beginning any test and after the entire test in the package. Run the following command:

```
nosetests test_module03.py -vs
```

Here is the output:

```
In setUpPackage()...
In setUpModule()...
Creating object : mymathlib
In setUpClass()...
test.test_module03.TestClass02.test_case01 ...
In setUp()...
In test_case01()
        In tearDown()...
        ok
        test.test_module03.TestClass02.test_case02 ...
        In setUp()...
        In test_case02()
        In tearDown()...
        ok

        In tearDownClass()...
        In tearDownModule()...
        Destroying object : mymathlib
        In tearDownPackage()...

        ----------------------------------------------------------------------
        Ran 2 tests in 0.012s

        OK
```

Alternate Names of the nose Fixtures

This table lists the alternate names of the nose fixtures.

Fixture	Alternative Name(s)
setUpPackage	setup, setUp, or setup_package
tearDownPackage	teardown, tearDown, or teardown_package
setUpModule	setup, setUp, or setup_module
tearDownModule	teardown, tearDown, or teardown_module
setUpClass	setupClass, setup_class, setupAll, or setUpAll
tearDownClass	teardownClass, teardown_class, teardownAll,or tearDownAll
setUp (class method fixtures)	setup
tearDown (class method fixtures)	teardown

assert_equals()

Until now, you have been using Python's built-in keyword assert to check the actual results against expected values. nose has its own assert_equals() method for this. The code in Listing 4-7 demonstrates the use of assert_equals() and assert.

Listing 4-7. test_module05.py

```
from nose.tools import assert_equals

def test_case01():
    print("In test_case01()...")
    assert 2+2 == 5

def test_case02():
    print("In test_case02()...")
    assert_equals(2+2, 5)
```

Run the code in Listing 4-7. The following shows the output:

```
In setUpPackage()...
test.test_module05.test_case01 ... In test_case01()...
FAIL
test.test_module05.test_case02 ... In test_case02()...
FAIL
In tearDownPackage()...

======================================================================
FAIL: test.test_module05.test_case01
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/usr/local/lib/python3.4/dist-packages/nose/case.py", line 198,
  in runTest
    self.test(*self.arg)
  File "/home/pi/book/code/chapter04/test/test_module05.py", line 6,
  in test_case01
    assert 2+2 == 5
AssertionError

======================================================================
FAIL: test.test_module05.test_case02
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/usr/local/lib/python3.4/dist-packages/nose/case.py", line 198,
  in runTest
    self.test(*self.arg)
  File "/home/pi/book/code/chapter04/test/test_module05.py", line 11,
  in test_case02
    assert_equals(2+2, 5)
AssertionError: 4 != 5

----------------------------------------------------------------------
Ran 2 tests in 0.013s

FAILED (failures=2)
```

Both the test cases failed due to incorrect test inputs. Note the difference between the logs printed by these test methods. In `test_case02()`, you get more information about the cause of the failure, as you are using nose's `assert_equals()` method.

Testing Tools

`nose.tools` has a few methods and decorators that come in very handy while you're automating tests.

This section looks at a few of those testing tools.

ok_ and eq_

`ok_` and `eq_` are shorthand for `assert` and `assert_equals()`, respectively. They also come with a parameter for an error message when the test case fails. The code in Listing 4-8 demonstrates this.

Listing 4-8. test_module06.py

```
from nose.tools import ok_, eq_

def test_case01():
    ok_(2+2 == 4, msg="Test Case Failure...")

def test_case02():
    eq_(2+2, 4, msg="Test Case Failure...")

def test_case03():
    ok_(2+2 == 5, msg="Test Case Failure...")

def test_case04():
    eq_(2+2, 5, msg="Test Case Failure...")
```

The following shows the output of the code in Listing 4-8.

```
In setUpPackage()...
test.test_module06.test_case01 ... ok
test.test_module06.test_case02 ... ok
test.test_module06.test_case03 ... FAIL
test.test_module06.test_case04 ... FAIL
In tearDownPackage()...
```

```
======================================================================
FAIL: test.test_module06.test_case03
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/usr/local/lib/python3.4/dist-packages/nose/case.py", line 198,
  in runTest
    self.test(*self.arg)
  File "/home/pi/book/code/chapter04/test/test_module06.py", line 13,
  in test_case03
    ok_(2+2 == 5, msg="Test Case Failure...")
AssertionError: Test Case Failure...

======================================================================
FAIL: test.test_module06.test_case04
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/usr/local/lib/python3.4/dist-packages/nose/case.py", line 198,
  in runTest
    self.test(*self.arg)
  File "/home/pi/book/code/chapter04/test/test_module06.py", line 17,
  in test_case04
    eq_(2+2, 5, msg="Test Case Failure...")
AssertionError: Test Case Failure...

----------------------------------------------------------------------
Ran 4 tests in 0.015s

FAILED (failures=2)
```

The @raises() Decorator

When you use `raises` decorator before the test, it must raise one of the exceptions mentioned in the list of exceptions associated with the `@raises()` decorator. Listing 4-9 demonstrates this idea.

Listing 4-9. test_module07.py

```
from nose.tools import raises

@raises(TypeError, ValueError)
def test_case01():
    raise TypeError("This test passes")

@raises(Exception)
def test_case02():
    pass
```

The output is as follows:

```
In setUpPackage()...
test.test_module07.test_case01 ... ok
test.test_module07.test_case02 ... FAIL
In tearDownPackage()...

======================================================================
FAIL: test.test_module07.test_case02
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/usr/local/lib/python3.4/dist-packages/nose/case.py", line 198, in
runTest
    self.test(*self.arg)
  File "/usr/local/lib/python3.4/dist-packages/nose/tools/nontrivial.py",
line 67, in newfunc
    raise AssertionError(message)
AssertionError: test_case02() did not raise Exception

----------------------------------------------------------------------
Ran 2 tests in 0.012s

FAILED (failures=1)
```

As you can see, `test_case02()` fails as it does not raise a exception when it is supposed to. You can cleverly use this to write negative test cases.

The @timed() decorator

If you are using a timed decorator with the test, then the test must finish within the time mentioned in the `@timed()` decorator to pass. The code in Listing 4-10 demonstrates that idea.

Listing 4-10. test_module10.py

```
from nose.tools import timed
import time

@timed(.1)
def test_case01():
    time.sleep(.2)
```

This test fails, as it takes more time to finish the execution of the test than is allotted in the `@timed()` decorator. The output of execution is as follows:

```
In setUpPackage()...
test.test_module08.test_case01 ... FAIL
In tearDownPackage()...
```

```
======================================================================
FAIL: test.test_module08.test_case01
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/usr/local/lib/python3.4/dist-packages/nose/case.py", line 198, in
runTest
    self.test(*self.arg)
  File "/usr/local/lib/python3.4/dist-packages/nose/tools/nontrivial.py",
line 100, in newfunc
    raise TimeExpired("Time limit (%s) exceeded" % limit)
nose.tools.nontrivial.TimeExpired: Time limit (0.1) exceeded

----------------------------------------------------------------------
Ran 1 test in 0.211s

FAILED (failures=1)
```

It is the collection or group of related tests that can be executed or scheduled to be executed together.

Report Generation

Let's look at the various ways to generate comprehensible reports using nose.

Creating an XML Report

nose has a built-in feature for generating XML reports. These are xUnit-style formatted reports. You have to use --with-xunit for generating the report. The report is generated in the current working directory.

Run the following command in the test directory:

```
nosetests test_module01.py -vs --with-xunit
```

The output will be as follows:

```
In setUpPackage()...
test.test_module01.test_case01 ... ok
In tearDownPackage()...

----------------------------------------------------------------------
XML: /home/pi/book/code/chapter04/test/nosetests.xml
----------------------------------------------------------------------
Ran 1 test in 0.009s

OK
```

The generated XML file is shown in Listing 4-11.

Listing 4-11. nosetests.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuite name="nosetests" tests="1" errors="0" failures="0" skip="0">
<testcase classname="test.test_module01" name="test_case01" time="0.002">
</testcase>
</testsuite>
```

Creating an HTML Report

nose does not have a built-in provision for HTML reports. You have to install a plugin for that. Run the following command to install the HTML output plugin:

```
sudo pip3 install nose-htmloutput
```

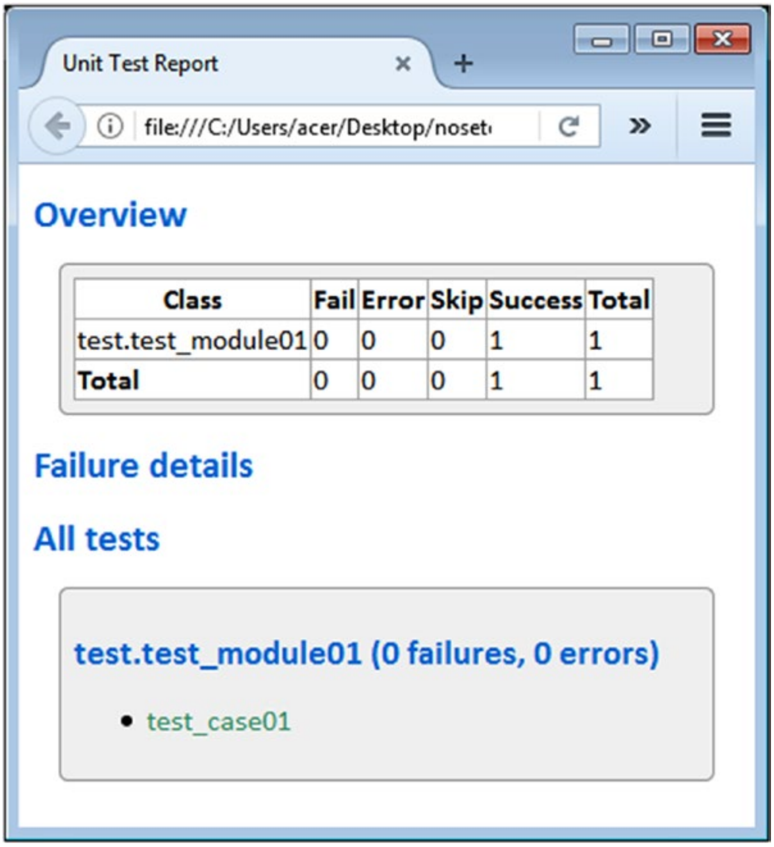
Once the plugin is installed, you can run the following command to execute the test:

```
nosetests test_module01.py -vs --with-html
```

Here is the output:

```
In setUpPackage()...
test.test_module01.test_case01 ... ok
In tearDownPackage()...

----------------------------------------------------------------------
HTML: nosetests.html
----------------------------------------------------------------------
Ran 1 test in 0.009s

OK
```

The plugin saves the output in the current location in a file called `nosetests.html`. Figure 4-3 shows a snapshot of the `nosetests.html` file, opened in a web browser.



Overview

Class	Fail	Error	Skip	Success	Total
test.test_module01	0	0	0	1	1
Total	0	0	0	1	1

Failure details

All tests

test.test_module01 (0 failures, 0 errors)

- test_case01

Figure 4-3. *The nosetests.html file*

Creating Color Output in the Console

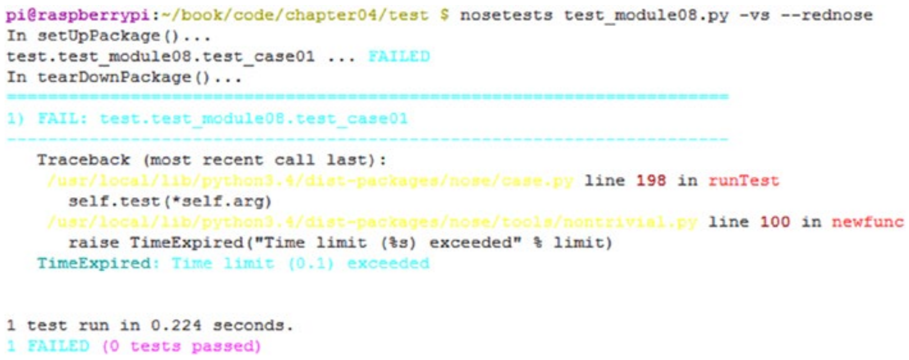
Until now, you saw the methods that generate formatted output files. While running `nosetest`, you must have observed that the console output is monochrome (white text on a dark background and vice versa). The plugin called `rednose` is used to create colored console output. You can install that plugin using the following command:

```
sudo pip3 install rednose
```

Once the plugin is installed, run the following command:

```
nosetests test_module08.py -vs --rednose
```

Figure 4-4 shows a screenshot of the output, although you won't see it in color here, due to the grayscale nature of the published book.



```
pi@raspberrypi:~/book/code/chapter04/test $ nosetests test_module08.py -vs --rednose
In setUpPackage()...
test.test_module08.test_case01 ... FAILED
In tearDownPackage()...
======================================================================
1) FAIL: test.test_module08.test_case01
----------------------------------------------------------------------
   Traceback (most recent call last):
    /usr/local/lib/python3.4/dist-packages/nose/case.py line 198 in runTest
      self.test(*self.arg)
    /usr/local/lib/python3.4/dist-packages/nose/tools/nontrivial.py line 100 in newfunc
      raise TimeExpired("Time limit (%s) exceeded" % limit)
   TimeExpired: Time limit (0.1) exceeded


1 test run in 0.224 seconds.
1 FAILED (0 tests passed)
```

Figure 4-4. *A rednose demo*

Running unittest Tests from nose

In the beginning of the chapter, you read that you can run `unittest` tests with `nose`. Let's try that now. Navigate to the `chapter03` directory. Run the following command to discover and execute all of the `unittest` tests automatically:

```
nosetests -v
```

This will be the output:

```
test_case01 (test.test_module01.TestClass01) ... ok
test_case02 (test.test_module01.TestClass01) ... ok
test_case01 (test.test_module02.TestClass02) ... ok
test_case02 (test.test_module02.TestClass02) ... ok
test_case01 (test.test_module03.TestClass03) ... ok
test_case02 (test.test_module03.TestClass03) ... ok
test_case03 (test.test_module03.TestClass03) ... FAIL
test_case04 (test.test_module03.TestClass03) ... FAIL
test_case01 (test.test_module04.TestClass04) ... ok
```

I am truncating the output as it would otherwise fill a lot of pages. Run the command yourself to see the entire output.

Advantages of nose over unittest

Here is a summary of the advantages of `nose` over `unittest`:

- Unlike `unittest`, `nose` does not require you to extend test cases from a parent class. This results in less code.
- Using `nose`, you can write test functions. This is not possible in `unittest`.

- nose has more fixtures than unittest. In addition to the regular unittest fixtures, nose has package- and function-level fixtures.
- nose has alternate names for fixtures.
- nose.tools offers many features for automating test cases.
- Test discovery is simpler in nose than in unittest, as nose does not need a Python interpreter with the discover sub-command.
- nose can recognize and run unittest tests easily.

Disadvantages of nose

The only and the biggest disadvantage of nose is that it is not under active development and has been in maintenance mode for the past several years. It will likely cease without a new person or team to take over its maintenance. If you're planning to start a project and are looking for a suitable automation framework for Python 3, then pytest, nose2, or plain unittest.

You might be wondering why I even spent time covering nose if it is not being actively developed. The reason is that learning a more advanced framework like nose helps you understand the limitations of unittest. Also, if you are working with an older project that uses nose as the test automation and/or unit testing framework, it will help you understand your tests.

Using Nose 2

nose2 is the next generation of testing for Python. It is based on the plugins branch of unittest2.

nose2 aims to improve on nose as follows:

- It provides a better plugin API.
- It is easier for users to configure.
- It simplifies internal interfaces and processes.
- It supports Python 2 and 3 from the same codebase.
- It encourages greater community involvement in its development.
- Unlike nose, it is under active development.

It can be installed conveniently using the following command:

```
sudo pip3 install nose2
```

Once installed, nose2 can be invoked by running nose2 at the command prompt. It can be used to auto-discover and run the unittest and nose test modules. Run the nose2 -h command at the command prompt to get help with the various nose2 command-line options.

The following are the important differences between nose and nose2:

- Python versions

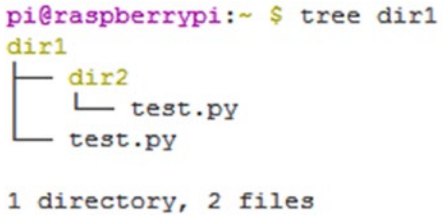
  nose supports Python version 2.4 and above. nose2 supports pypy, 2.6, 2.7, 3.2, 3.3, 3.4, and 3.5. nose2 does not support all the versions, as it is not possible to support all the Python versions in a single codebase.

- Test loading

  nose loads and executes test modules one by one, which is called *lazy loading*. On the contrary, nose2 loads all the modules first and then executes them all at once.

- Test discovery

  Because of the difference between the test loading techniques, nose2 does not support all the project layouts. The layout shown in Figure 4-5 is supported by nose. However, it will not be loaded correctly by nose2. nose can distinguish between ./dir1/test.py and ./dir1/dir2/test.py.



```
pi@raspberrypi:~ $ tree dir1
dir1
├── dir2
│   └── test.py
└── test.py

1 directory, 2 files
```

Figure 4-5. *nose2 unsupported test layout*

EXERCISE 4-1

Check if the codebase in your organization is using unittest, nose, or nose2. Consult with the owners of the codebase and plan a migration from these frameworks to a better and more flexible unit-testing framework.

Conclusion

In this chapter, you learned about the advanced unit testing framework nose. Unfortunately, it is not being developed actively so you need to use nose2 as a test-runner for nose tests. In the next chapter, you will learn about and explore an advanced test automation framework called py.test.

CHAPTER 5

pytest

In an earlier chapter, we explored nose, which is an advanced and better framework for Python testing. Unfortunately, nose has not been under active development for the past several years. That makes it an unsuitable candidate for a test framework when you want to choose something for a long-term project. Moreover, there are many projects that use unittest or nose or a combination of both. You definitely need a framework that has more features than unittest, and unlike nose, it should be under active development. nose2 is more of a test-runner for unittest and an almost defunct tool. You need a unit test framework that's capable of discovering and running tests written in unittest and nose. It should be advanced and must be under active development. The answer is pytest.

This chapter extensively explores a modern, advanced, and better test automation framework, called pytest. First, you'll learn how pytest offers traditional xUnit style fixtures and then you will explore the advanced fixtures offered by pytest.

Introduction to pytest

pytest is not a part of Python's standard library. We have to install it in order to use it, just like we installed nose and nose2. Let's see how we can install it for Python 3. pytest can be installed conveniently by running the following command on Windows:

```
pip install pytest
```

For Linux and MacOS, you install it using pip3 as follows:

```
sudo pip3 install pytest
```

This installs pytest for Python 3.

You can check the installed version by running the following command:

```
py.test --version
```

The output is as follows:

```
This is pytest version 3.0.4, imported from /usr/local/lib/python3.4/dist-
packages/pytest.py
```

A. Pajankar, *Python Unit Test Automation*, DOI 10.1007/978-1-4842-2677-3_5

Simple Test

Before you begin, create a directory called chapter05 in the code directory. Copy the mypackage directory as it is from the chapter04 directory. Create a directory called test in chapter05. Save all the code files for this chapter in the test directory.

Just like when using nose, writing a simple test is very easy. See the code in Listing 5-1 as an example.

Listing 5-1. test_module01.py

```
def test_case01():
        assert 'python'.upper() == 'PYTHON'
```

In Listing 5-1, we are importing pytest in the first line. test_case01() is the test function. Recall that assert is a Python built-in keyword. Also, just like with nose, we do not need to extend these tests from any class. This helps keep the code uncluttered.

Run the test module with the following command:

```
python3 -m pytest test_module01.py
```

The output is as follows:

```
============================ test session starts ====================
platform linux -- Python 3.4.2, pytest-3.0.4, py-1.4.31, pluggy-0.4.0
rootdir: /home/pi/book/code/chapter05/test, inifile:
collected 1 items

test_module01.py .

========================== 1 passed in 0.05 seconds =================
```

You can also use verbose mode:

```
python3 -m pytest -v test_module01.py
```

The output is as follows:

```
========================= test session starts ===========================
platform linux -- Python 3.4.2, pytest-3.0.4, py-1.4.31, pluggy-0.4.0 --
/usr/bin/python3
cachedir: .cache
rootdir: /home/pi/book/code/chapter05/test, inifile:
collected 1 items

test_module01.py::test_case01 PASSED

======================== 1 passed in 0.04 seconds ====================
```

Running Tests with the py.test Command

You can also run these tests with `pytest`'s own command, called `py.test`:

```
py.test test_module01.py
```

The output is as follows:

```
====================== test session starts =========================
platform linux -- Python 3.4.2, pytest-3.0.4, py-1.4.31, pluggy-0.4.0
rootdir: /home/pi/book/code/chapter05/test, inifile:
collected 1 items

test_module01.py .

====================== 1 passed in 0.04 seconds ====================
```

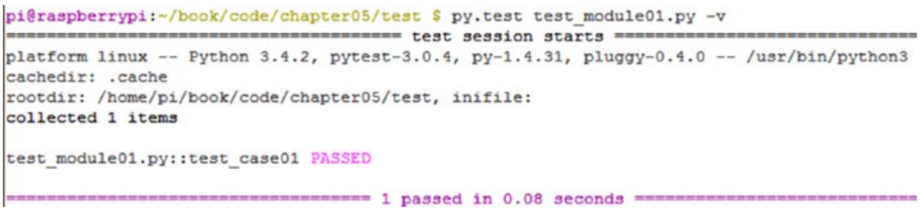
You can also use verbose mode as follows:

```
py.test test_module01.py -v
```

The output in the verbose mode is as follows:

```
=========================== test session starts ======================
platform linux -- Python 3.4.2, pytest-3.0.4, py-1.4.31, pluggy-0.4.0 --
/usr/bin/python3
cachedir: .cache
rootdir: /home/pi/book/code/chapter05/test, inifile:
collected 1 items

test_module01.py::test_case01 PASSED

=========================== 1 passed in 0.04 seconds =================
```

For the sake of simplicity and convenience, from now onward, we will use the same method to run these tests for rest of the chapter and book. We will use `pytest` in the next chapter to implement test-driven development. Also, observe when you run your own tests that the output of test execution is in color by default, although the book shows the results in black and white. You do not have to use any external or third-party plugin for this effect. Figure 5-1 shows a screenshot of an execution sample.



```
pi@raspberrypi:~/book/code/chapter05/test $ py.test test_module01.py -v
======================================== test session starts ================================
platform linux -- Python 3.4.2, pytest-3.0.4, py-1.4.31, pluggy-0.4.0 -- /usr/bin/python3
cachedir: .cache
rootdir: /home/pi/book/code/chapter05/test, inifile:
collected 1 items

test_module01.py::test_case01 PASSED

==================================== 1 passed in 0.08 seconds ============================
```

Figure 5-1. *Sample pytest execution*

Test Class and Test Package in pytest

Like all the previous test automation frameworks, in `pytest` you can create test classes and test packages. Take a look at the code in Listing 5-2 as an example.

Listing 5-2. test_module02.py

```
class TestClass01:

        def test_case01(self):
                assert 'python'.upper() == 'PYTHON'

        def test_case02(self):
                assert 'PYTHON'.lower() == 'python'
```

Also create an `__init__.py` file, as shown in Listing 5-3.

Listing 5-3. __init__.py

```
all = ["test_module01", "test_module02"]
```

Now navigate to the `chapter05` directory:

```
cd /home/pi/book/code/chapter05
```

And run the test package, as follows:

```
py.test test
```

You can see the output by running the previous command. You can also use the following command to run a test package in verbose mode.

```
py.test -v test
```

You can run a single test module within a package with the following command:

```
py.test -v test/test_module01.py
```

You can also run a specific test class as follows:

```
py.test -v test/test_module02.py::TestClass01
```

You can run a specific test method as follows:

```
py.test -v test/test_module02.py::TestClass01::test_case01
```

You can run a specific test function as follows:

```
py.test -v test/test_module01.py::test_case01
```

Test Discovery in pytest

pytest can discover and automatically run the tests, just like unittest, nose, and nose2 can. Run the following command in the project directory to initiate automated test discovery:

```
py.test
```

For verbose mode, run the following command:

```
py.test -v
```

xUnit-Style Fixtures

pytest has xUnit-style of fixtures. See the code in Listing 5-4 as an example.

Listing 5-4. test_module03.py

```
def setup_module(module):
    print("\nIn setup_module()...")

def teardown_module(module):
    print("\nIn teardown_module()...")

def setup_function(function):
    print("\nIn setup_function()...")

def teardown_function(function):
    print("\nIn teardown_function()...")

def test_case01():
    print("\nIn test_case01()...")

def test_case02():
    print("\nIn test_case02()...")

class TestClass02:

    @classmethod
    def setup_class(cls):
        print ("\nIn setup_class()...")

    @classmethod
    def teardown_class(cls):
        print ("\nIn teardown_class()...")

    def setup_method(self, method):
        print ("\nIn setup_method()...")

    def teardown_method(self, method):
        print ("\nIn teardown_method()...")

    def test_case03(self):
        print("\nIn test_case03()...")

    def test_case04(self):
        print("\nIn test_case04()...")
```

In this code, `setup_module()` and `teardown_module()` are module-level fixtures that are invoked before and after anything else in the module. `setup_class()` and `teardown_class()` are the class-level fixtures and they run before and after anything else in the class. You have to use the `@classmethod()` decorator with them. `setup_method()` and `teardown_method()` are method-level fixtures that run before and after every test method. `setup_function()` and `teardown_function()` are function-level fixtures that run before and after every test function in the module. In `nose`, you need the `@with_setup()` decorator with the test functions to assign those to the function level-fixtures. In `pytest`, function-level fixtures are assigned to all the test functions by default.

Also, just like with `nose`, you need to use the `-s` command-line option to see the detailed log on the command line.

Let's run the code with an additional `-s` option, as follows:

```
py.test -vs test_module03.py
```

Now, run the test again with the following command:

```
py.test -v test_module03.py
```

Compare the outputs of these modes of execution for a better understanding.

pytest Support for unittest and nose

pytest supports all the tests written in unittest and nose. pytest can automatically discover and run the tests written in unittest and nose. It supports all the xUnit-style fixtures for unittest test classes. It also supports most of the fixtures in nose. Try running py.test -v in the chapter03 and chapter04 directories.

Introduction to pytest Fixtures

Apart from supporting xUnit-style fixtures and unittest fixtures, pytest has its own set of fixtures that are flexible, extensible, and modular. This is one of the core strengths of pytest and why it's a popular choice of automation testers.

In pytest, you can create a fixture and use it as a resource where it is needed. Consider the code in Listing 5-5 as an example.

Listing 5-5. test_module04.py

```
import pytest

@pytest.fixture()
def fixture01():
    print("\nIn fixture01()...")

def test_case01(fixture01):
    print("\nIn test_case01()...")
```

In Listing 5-5, fixture01() is the fixture function. It is because we are using the @pytest.fixture() decorator with that. test_case01() is a test function that uses fixture01(). For that, we are passing fixture01 as an argument to test_case01().

Here is the output:

```
============================ test session starts =====================
platform linux -- Python 3.4.2, pytest-3.0.4, py-1.4.31, pluggy-0.4.0 --
/usr/bin/python3
cachedir: .cache
rootdir: /home/pi/book/code/chapter05/test, inifile:
collected 1 items

test_module04.py::test_case01
In fixture01()...

In test_case01()...
PASSED

========================== 1 passed in 0.04 seconds ===================
```

As you can see, `fixture01()` is invoked before the test function `test_case01()`. You could also use the `@pytest.mark.usefixtures()` decorator, which achieves the same result. The code in Listing 5-6 is implemented with this decorator and it produces the same output as Listing 5-5.

Listing 5-6. test_module05.py

```
import pytest

@pytest.fixture()
def fixture01():
    print("\nIn fixture01()...")

@pytest.mark.usefixtures('fixture01')
def test_case01(fixture01):
    print("\nIn test_case01()...")
```

The output of Listing 5-6 is exactly the same as the code in Listing 5-5.

You can use the `@pytest.mark.usefixtures()` decorator for a class, as shown in Listing 5-7.

Listing 5-7. test_module06.py

```
import pytest

@pytest.fixture()
def fixture01():
    print("\nIn fixture01()...")

@pytest.mark.usefixtures('fixture01')
class TestClass03:
    def test_case01(self):
        print("I'm the test_case01")

    def test_case02(self):
        print("I'm the test_case02")
```

Here is the output:

```
============================ test session starts =======================
platform linux -- Python 3.4.2, pytest-3.0.4, py-1.4.31, pluggy-0.4.0 --
/usr/bin/python3
cachedir: .cache
rootdir: /home/pi/book/code/chapter05/test, inifile:
collected 2 items

test_module06.py::TestClass03::test_case01
In fixture01()...
```

```
I'm the test_case01
PASSED
test_module06.py::TestClass03::test_case02
In fixture01()...
I'm the test_case02
PASSED

========================== 2 passed in 0.08 seconds ====================
```

If you want to run a block of code after the test with a fixture has run, you have to add a finalizer function to the fixture. Listing 5-8 demonstrates this idea.

Listing 5-8. test_module07.py

```
import pytest

@pytest.fixture()
def fixture01(request):
    print("\nIn fixture...")

    def fin():
        print("\nFinalized...")
    request.addfinalizer(fin)

@pytest.mark.usefixtures('fixture01')
def test_case01():
    print("\nI'm the test_case01")
```

The output is as follows:

```
=========================== test session starts ========================
platform linux -- Python 3.4.2, pytest-3.0.4, py-1.4.31, pluggy-0.4.0 --
/usr/bin/python3
cachedir: .cache
rootdir: /home/pi/book/code/chapter05/test, inifile:
collected 1 items

test_module07.py::test_case01
In fixture...

I'm the test_case01
PASSED
Finalized...

========================= 1 passed in 0.05 seconds =====================
```

pytest provides access to the fixture information on the requested object. Listing 5-9 demonstrates this concept.

Listing 5-9. test_module08.py

```
import pytest

@pytest.fixture()
def fixture01(request):
    print("\nIn fixture...")
    print("Fixture Scope: " + str(request.scope))
    print("Function Name: " + str(request.function.__name__))
    print("Class Name: " + str(request.cls))
    print("Module Name: " + str(request.module.__name__))
    print("File Path: " + str(request.fspath))

@pytest.mark.usefixtures('fixture01')
def test_case01():
    print("\nI'm the test_case01")
```

The following is the output of Listing 5-9:

```
============================ test session starts =======================
platform linux -- Python 3.4.2, pytest-3.0.4, py-1.4.31, pluggy-0.4.0 --
/usr/bin/python3
cachedir: .cache
rootdir: /home/pi/book/code/chapter05/test, inifile:
collected 1 items

test_module08.py::test_case01
In fixture...
Fixture Scope: function
Function Name: test_case01
Class Name: None
Module Name: test.test_module08
File Path: /home/pi/book/code/chapter05/test/test_module08.py

I'm the test_case01
PASSED

========================= 1 passed in 0.06 seconds =====================
```

Scope of pytest Fixtures

pytest provides you with a set of scope variables to define exactly when you want to use the fixture. The default scope of any fixture is the function level. It means that, by default, the fixtures are at the level of function.

The following is the list of scopes for `pytest` fixtures:

- `function`: Runs once per test
- `class`: Runs once per class of tests
- `module`: Runs once per module
- `session`: Runs once per session

To use these, define them like this:

```
@pytest.fixture(scope="class")
```

- Use the `function` scope if you want the fixture to run after every single test. This is fine for smaller fixtures.
- Use the `class` scope if you want the fixture to run in each class of tests. Typically, you'll group tests that are alike in a class, so this may be a good idea, depending on how you structure things.
- Use the `module` scope if you want the fixture to run at the start of the current file and then after the file has finished its tests. This can be good if you have a fixture that accesses the database and you set up the database at the beginning of the module and then the finalizer closes the connection.
- Use the `session` scope if you want to run the fixture at the first test and run the finalizer after the last test has run.

There is no scope for packages in `pytest`. However, you can cleverly use the `session` scope as a package-level scope by making sure that only a specific test package runs in a single session.

pytest.raises()

In `unittest`, you have `assertRaises()` to check if any test raises an exception. There is a similar method in `pytest`. It is implemented as `pytest.raises()` and is useful for automating negative test scenarios.

Consider the code shown in Listing 5-10.

Listing 5-10. test_module09.py

```
import pytest

def test_case01():
    with pytest.raises(Exception):
        x = 1 / 0

def test_case02():
    with pytest.raises(Exception):
        x = 1 / 1
```

In Listing 5-10, the line with `pytest.raises(Exception)` checks if an exception is raised in the code. If an exception is raised in the block of the code that include the exception, the test passes; otherwise, it fails.

Here is Listing 5-10's output:

```
========================= test session starts ==============================
platform linux -- Python 3.4.2, pytest-3.0.4, py-1.4.31, pluggy-0.4.0 --
/usr/bin/python3
cachedir: .cache
rootdir: /home/pi/book/code/chapter05/test, inifile:
collected 2 items

test_module09.py::test_case01 PASSED
test_module09.py::test_case02 FAILED

============================= FAILURES =====================================
_____________________________ test_case02 _________________________________

    def test_case02():
        with pytest.raises(Exception):
>           x = 1 / 1
E           Failed: DID NOT RAISE <class 'Exception'>

test_module09.py:10: Failed
==================== 1 failed, 1 passed in 0.21 seconds ===================
```

In `test_case01()`, an exception is raised, so it passes. `test_case02()` does not raise any exception, so it fails. As mentioned earlier, this is extremely useful for testing negative scenarios.

Important pytest Command-Line Options

Some of pytest's more important command-line options are discussed in the following sections.

Help

For help, run `py.test -h`. It will display a list and usage of various command-line options.

Stopping After the First (or N) Failures

You can stop the execution of tests after the first failure using `py.test -x`. In the same way, you can use `py.test --maxfail=5` to stop execution after five failures. You can also change the argument provided to `--maxfail`.

Profiling Test Execution Duration

You can use the `py.test --durations=10` command to show the 10 slowest tests. You can change the argument provided to `--duration`. Try running this command on the `chapter05` directory as an example.

JUnit-Style Logs

You can generate JUnit-style XML log files by running the following command:

```
py.test --junitxml=result.xml
```

The XML file will be generated in the current directory.

Generating a Plain Result

In an earlier section, you learned how to generate an XML log file. In same way, you can also generate a plaintext result file by running the following command:

```
py.test --resultlog=result.log
```

The plaintext log file will be generated in the current directory.

Sending a Test Report to Online pastebin Service

The following command sends the entire execution log to an online remote `pastebin` service:

```
py.test -v --pastebin=all
```

As of now, only pasting to the `http://bpaste.net` service is implemented. The output of the execution will contain a web link where the log has been stored. Open the link in a web browser to view the log. This is a great way to share the log across a geographically distributed team. Note that the link to the online `pastebin` log page expires in seven days.

Conclusion

The following are the reasons I use `pytest` and recommend that all Python enthusiasts and professionals use it:

- It is better than `unittest`. The resulting code is cleaner and simpler.
- Unlike with `nose`, `pytest` is still under active development.

- It has great features for controlling the test execution.
- It can generate XML as well as plaintext results without an additional plugin.
- It can run `unittest` tests.
- It has its own set of advanced fixtures that are modular in nature.

If you are working on a project where they use `unittest`, `nose`, or `doctest` as the test framework for Python, I recommend migrating your tests to `pytest`.

CHAPTER 6

Tips and Tricks

In the first chapter of the book, you about learned the history and philosophy of Python. Subsequent chapters explored the features of various test automation frameworks in Python. The frameworks you explored included `doctest`, `unittest`, `nose`, `nose2`, and `pytest`. This chapter looks at coding conventions that will make the test discovery easier across the frameworks. Then, we will look at the concept of *test-driven development* and how it can be implemented in Python 3 projects with the help of `pytest`.

Coding and Filenaming Conventions for Easier Test Discovery

You have seen that all the `xUnit`-style frameworks have the feature of test discovery, that is, the automated detection, execution, and report generation of tests. This is a very important feature, as it makes life easier for code testers. You can even schedule the test discovery process by using OS schedulars (for example, `cron` in Linux-based operating systems and Windows Schedular in Windows), and they will run tests at scheduled times automatically.

In order to ensure that the test discovery system detects all the tests successfully, I usually follow these code and filename conventions:

- Names of all the test modules (the test files) should start with `test_`
- Names of all the test functions should start with `test_`
- Names of all the test classes should start with `Test`
- Names of all the test methods should start with `test_`
- Group all the tests into test classes and packages
- All the packages with test code should have an `__init__.py` file

It is always a good idea to follow the PEP 8 convention for the code. It can be found at `https://www.python.org/dev/peps/pep-0008/`.

If you use these conventions for your code and filenames, the test discovery feature of all the test automation frameworks—including `unittest`, `nose`, `nose2`, and `pytest`—will detect the tests without any problem. So, the next time you write your tests, follow these conventions for best results.

Test-Driven Development with pytest

Test-driven development (TDD) is a paradigm whereby you implement the new feature or requirement by writing the tests first, watch them fail, and then write the code to make the failed tests pass. Once the basic skeleton of the features is implemented this way, you then build on this by altering the tests and then changing the development code to accommodate the added functionality. You repeat this process as many times as needed to accommodate all new requirements.

Essentially, TDD is a cycle where you write the tests first, watch them fail, implement the required features, and repeat this process until the new features are added to the existing code.

By writing the automated tests before the development code, it forces you to think about the problem at hand first. As you start to build your tests, you have to think about the way you write the development code that must pass the already-written automated tests in order to be accepted.

Figure 6-1 sums up the TDD approach.

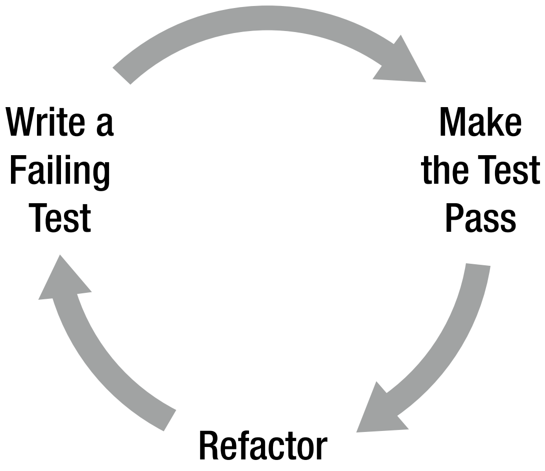


Figure 6-1. *TDD flow*

To see how TDD is implemented in Python with `pytest`, create a directory called `chapter06` for this TDD in the `code` directory. You will use this directory for the TDD exercise.

Create the test module shown in Listing 6-1 in the `chapter06` directory.

Listing 6-1. test_module01.py

```
class TestClass01:

        def test_case01(self):
               calc = Calculator()
               result = calc.add(2, 2)
               assert 4 == result
```

Run the code in Listing 6-1 with the following command:

```
py.test -vs test_module01.py
```

The output will be as follows:

```
============================ test session starts =============================
platform linux -- Python 3.4.2, pytest-3.0.4, py-1.4.31, pluggy-0.4.0 -- /
usr/bin/python3
cachedir: .cache
rootdir: /home/pi/book/code/chapter06, inifile:
collected 1 items

test_module01.py::TestClass01::test_case01 FAILED

================================= FAILURES ==================================
__________________________ TestClass01.test_case01 __________________________

self = <test_module01.TestClass01 object at 0x763c03b0>

    def test_case01(self):
>           calc = Calculator()
E           NameError: name 'Calculator' is not defined

test_module01.py:4: NameError
========================== 1 failed in 0.29 seconds =========================
```

From this output, you can see that the problem is due to us not importing `Calculator`. That is because we have not created the `Calculator` module yet! So let's define the `Calculator` module in a file called `calculator.py`, as shown in Listing 6-2, under the same directory,

Listing 6-2. calculator.py

```
class Calculator:

        def add(self, x, y):
               pass
```

Make sure that there are no errors in `calculator.py` by running the following command every time you modify the module:

```
python3 calculator.py
```

Now import `Calculator` in the test module, as shown in Listing 6-3.

Listing 6-3. test_module01.py

```
from calculator import Calculator

class TestClass01:

        def test_case01(self):
               calc = Calculator()
               result = calc.add(2, 2)
               assert 4 == result
```

Run the `test_module01.py` again. The output will be as follows:

```
=========================== test session starts ==============================
platform linux -- Python 3.4.2, pytest-3.0.4, py-1.4.31, pluggy-0.4.0 --
/usr/bin/python3
cachedir: .cache
rootdir: /home/pi/book/code/chapter06, inifile:
collected 1 items

test_module01.py::TestClass01::test_case01 FAILED

================================ FAILURES ==================================
__________________________ TestClass01.test_case01 __________________________

self = <test_module01.TestClass01 object at 0x762c24b0>

    def test_case01(self):
            calc = Calculator()
            result = calc.add(2, 2)
>           assert 4 == result
E           assert 4 == None

test_module01.py:9: AssertionError
========================== 1 failed in 0.32 seconds ==========================
```

The add() method returns the wrong value (i.e., pass), as it does not do anything at the moment. Fortunately, pytest returns the line with the error in the test run so you can decide what you need to change. Let's fix the code in the add() method in calculator.py as shown in Listing 6-4.

Listing 6-4. calculator.py

```
class Calculator:

        def add(self, x, y):
               return x+y
```

You can run the test module again. Here is the output:

```
============================ test session starts ============================
platform linux -- Python 3.4.2, pytest-3.0.4, py-1.4.31, pluggy-0.4.0 --
/usr/bin/python3
cachedir: .cache
rootdir: /home/pi/book/code/chapter06, inifile:
collected 1 items

test_module01.py::TestClass01::test_case01 PASSED

========================= 1 passed in 0.08 seconds ==========================
```

Now you can add more test cases to the test module (as shown in Listing 6-5) to check for more features,

Listing 6-5. test_module01.py

```
from calculator import Calculator
import pytest

class TestClass01:

        def test_case01(self):
                calc = Calculator()
                result = calc.add(2, 2)
                assert 4 == result

        def test_case02(self):
                with pytest.raises(ValueError):
                        result = Calculator().add(2, 'two')
```

In the modified code shown in Listing 6-5, we're trying to add an integer and a string, which should raise a ValueError exception.

If you run the modified test module, you get the following:

```
============================ test session starts =============================
platform linux -- Python 3.4.2, pytest-3.0.4, py-1.4.31, pluggy-0.4.0 -- /
usr/bin/python3
cachedir: .cache
rootdir: /home/pi/book/code/chapter06, inifile:
collected 2 items

test_module01.py::TestClass01::test_case01 PASSED
test_module01.py::TestClass01::test_case02 FAILED

================================ FAILURES ====================================
_________________________ TestClass01.test_case02 ___________________________

self = <test_module01.TestClass01 object at 0x7636f050>

    def test_case02(self):
            with pytest.raises(ValueError):
>                   result = Calculator().add(2, 'two')

test_module01.py:14:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

self = <calculator.Calculator object at 0x7636faf0>, x = 2, y = 'two'

    def add(self, x, y):
>           return x+y
E           TypeError: unsupported operand type(s) for +: 'int' and 'str'

calculator.py:4: TypeError
==================== 1 failed, 1 passed in 0.33 seconds ======================
```

As you can see in the output, the second test fails as it does not detect a ValueError exception. So, let's add the provision to check if both the arguments are numeric, else raise a ValueError exception—see Listing 6-6.

Listing 6-6. calculator.py

```
class Calculator:

        def add(self, x, y):
                number_types = (int, float, complex)

               if isinstance(x, number_types) and isinstance(y, number_types):
                       return x + y
                else:
                       raise ValueError
```

Finally, Listing 6-7 shows how to add two more test cases to the test module to check if `add()` is behaving as expected.

Listing 6-7. test_module01.py

```
from calculator import Calculator
import pytest

class TestClass01:

        def test_case01(self):
                calc = Calculator()
                result = calc.add(2, 2)
                assert 4 == result

        def test_case02(self):
                with pytest.raises(ValueError):
                        result = Calculator().add(2, 'two')

        def test_case03(self):
                with pytest.raises(ValueError):
                        result = Calculator().add('two', 2)

        def test_case04(self):
                with pytest.raises(ValueError):
                        result = Calculator().add('two', 'two')
```

When you run the test module in Listing 6-7, you will get the following output:

```
============================ test session starts ============================
platform linux -- Python 3.4.2, pytest-3.0.4, py-1.4.31, pluggy-0.4.0 -- /
usr/bin/python3
cachedir: .cache
rootdir: /home/pi/book/code/chapter06, inifile:
collected 4 items

test_module01.py::TestClass01::test_case01 PASSED
test_module01.py::TestClass01::test_case02 PASSED
test_module01.py::TestClass01::test_case03 PASSED
test_module01.py::TestClass01::test_case04 PASSED

========================= 4 passed in 0.14 seconds ==========================
```

This is how TDD is implemented in real-life projects. You write a failing test first, refactor the development code, and continue the same process until the test passes. When you want to add a new feature, you repeat this process to implement it.