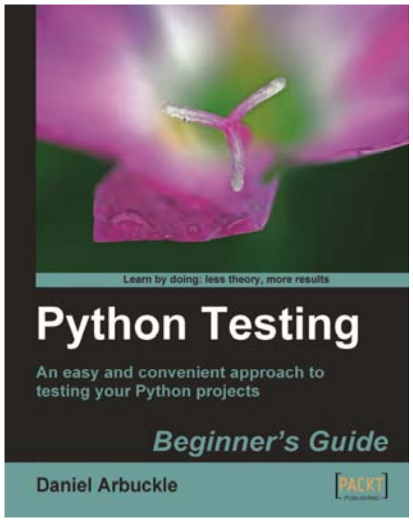
Python Testing Beginner's Guide

Daniel Arbuckle



Chapter No.5

"When Doctest isn't Enough: Unittest to the Rescue"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.5" Running Your Tests: Follow Your Nose"

A synopsis of the book's content

Information on where to buy this book

About the Author

Daniel Arbuckle received his Ph. D. in computer science from the University of Southern California in 2007. He is an active member of the Python community and an avid unit tester.

I would like to thank Grig, Titus, and my family for their companionship and encouragement along the way.

Python Testing Beginner's Guide:

Like any programmer, you need to be able to produce reliable code that conforms to a specification, which means that you need to test your code. In this book, you'll learn how to use techniques and Python tools that reduce the effort involved in testing, and at the same time make it more useful—and even fun.

You'll learn about several of Python's automated testing tools, and you'll learn about the philosophies and methodologies that they were designed to support, like unit testing and test-driven development. When you're done, you'll be able to produce thoroughly tested code faster and more easily than ever before, and you'll be able to do it in a way that doesn't distract you from your "real" programming.

What This Book Covers

Chapter 1: *Testing for Fun and Profit* introduces Python test-driven development and various testing methods.

Chapter 2: *Doctest: The Easiest Testing Tool* covers the doctest tool and teaches you how to use it.

Chapter 3: *Unit Testing with Doctest* introduces the ideas of unit testing and test-driven development, and applies doctest to create unit tests.

Chapter 4: *Breaking Tight Coupling by using Mock Objects* covers mock objects and the Python Mocker tool.

Chapter 5: *When Doctest isn't Enough: Unittest to the Rescue* introduces the unittest framework and discusses when it is preferred over doctest.

Chapter 6: *Running Your Tests: Follow Your Nose* introduces the Nose test runner, and discusses project organization. Chapter 7: *Developing a Test-Driven Project* walks through a complete test-driven development process.

Chapter 8: *Testing Web Application Frontends using Twill* applies the knowledge gained from previous chapters to web applications, and introduces the Twill tool.

Chapter 9: *Integration Testing and System Testing* teaches how to build from unit tests to tests of a complete soft ware system.

Chapter 10: *Other Testing Tools and Techniques* introduces code coverage and continuous integration, and teaches how to tie automated testing into version control systems.

Appendix: *Answers to Pop Quizes* contains the answers to all pop quizes, chapter-wise.

5
When Doctest isn't Enough: Unittest to the Rescue

As the tests get more detailed (or complex), or they require more setup code to prepare the way for them, doctest begins to get a little bit annoying. The very simplicity that makes it the best way to write testable specifications and other simple tests starts to interfere with writing tests for complicated things.

In this chapter we shall:

- Learn how to write and execute tests in the unittest framework
- Learn how to express familiar testing concepts using unittest
- Discuss the specific features that make unittest suitable for more complicated testing scenarios
- Learn about of couple of Mocker's features that integrate well with unittest

So let's get on with it!

Basic unittest

Before we start talking about new concepts and features, let's take a look at how to use unittest to express the ideas that we've already learned about. That way, we'll have something solid to ground our new understanding into.

Time for action – testing PID with unittest

We'll revisit the PID class (or at least the tests for the PID class) from Chapter 3. We'll rewrite the tests so that they operate within the unittest framework.

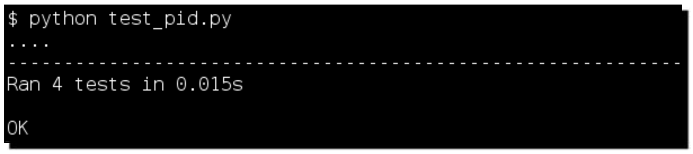
Before moving on, take a moment to refer back to the final version of the `pid.txt` file from Chapter 3. We'll be implementing the same tests using the unittest framework.

1. Create a new file called `test_pid.py` in the same directory as `pid.py`. Notice that this is a `.py` file: unittest tests are pure python source code, rather than being plain text with source code embedded in it. That means the tests will be less useful from a documentary point of view, but grants other benefits in exchange.
2. Insert the following code into your newly-created `test_pid.py` (and please note that a few lines are long enough to get wrapped on the book's page):

```
from unittest import TestCase, main
from mocker import Mocker

import pid

class test_pid_constructor(TestCase):
    def test_without_when(self):
        mocker = Mocker()
        mock_time = mocker.replace('time.time')
        mock_time()
        mocker.result(1.0)

        mocker.replay()

        controller = pid.PID(P=0.5, I=0.5, D=0.5,
                             setpoint=0, initial=12)

        mocker.restore()
        mocker.verify()

        self.assertEqual(controller.gains, (0.5, 0.5, 0.5))
        self.assertAlmostEqual(controller.setpoint[0], 0.0)
        self.assertEqual(len(controller.setpoint), 1)
        self.assertAlmostEqual(controller.previous_time, 1.0)
        self.assertAlmostEqual(controller.previous_error, -12.0)
        self.assertAlmostEqual(controller.integrated_error, 0)

    def test_with_when(self):
        controller = pid.PID(P=0.5, I=0.5, D=0.5,
                             setpoint=1, initial=12,
                             when=43)

        self.assertEqual(controller.gains, (0.5, 0.5, 0.5))
        self.assertAlmostEqual(controller.setpoint[0], 1.0)
        self.assertEqual(len(controller.setpoint), 1)
        self.assertAlmostEqual(controller.previous_time, 43.0)
        self.assertAlmostEqual(controller.previous_error, -11.0)
        self.assertAlmostEqual(controller.integrated_error, 0)

class test_calculate_response(TestCase):
    def test_without_when(self):
        mocker = Mocker()
        mock_time = mocker.replace('time.time')
        mock_time()
        mocker.result(1.0)
        mock_time()
        mocker.result(2.0)
        mock_time()
        mocker.result(3.0)
        mock_time()
        mocker.result(4.0)
        mock_time()
        mocker.result(5.0)

        mocker.replay()

        controller = pid.PID(P=0.5, I=0.5, D=0.5,
                             setpoint=0, initial=12)

        self.assertEqual(controller.calculate_response(6), -3)
        self.assertEqual(controller.calculate_response(3), -4.5)
        self.assertEqual(controller.calculate_response(-1.5), -0.75)
        self.assertEqual(controller.calculate_response(-2.25),
-1.125)

        mocker.restore()
        mocker.verify()
    def test_with_when(self):
        controller = pid.PID(P=0.5, I=0.5, D=0.5,
                             setpoint=0, initial=12,
                             when=1)

        self.assertEqual(controller.calculate_response(6, 2), -3)
        self.assertEqual(controller.calculate_response(3, 3), -4.5)
        self.assertEqual(controller.calculate_response(-1.5, 4),
-0.75)
        self.assertEqual(controller.calculate_response(-2.25, 5),
-1.125)

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

3. Run the tests by typing:

   $ python test_pid.py



```
$ python test_pid.py
....
----------------------------------------------------------
Ran 4 tests in 0.015s

OK
```

What just happened?

Let's go through the code section and see what each part does. After that, we'll talk about what it all means when put together.

```
from unittest import TestCase, main
from mocker import Mocker
import pid
class test_pid_constructor(TestCase):
    def test_without_when(self):
        mocker = Mocker()
        mock_time = mocker.replace('time.time')
        mock_time()
        mocker.result(1.0)
        mocker.replay()
        controller = pid.PID(P=0.5, I=0.5, D=0.5,
                             setpoint=0, initial=12)
        mocker.restore()
        mocker.verify()
        self.assertEqual(controller.gains, (0.5, 0.5, 0.5))
        self.assertAlmostEqual(controller.setpoint[0], 0.0)
        self.assertEqual(len(controller.setpoint), 1)
        self.assertAlmostEqual(controller.previous_time, 1.0)
        self.assertAlmostEqual(controller.previous_error, -12.0)
        self.assertAlmostEqual(controller.integrated_error, 0)
```

After a little bit of setup code, we have a test that the PID controller works correctly when not given a `when` parameter. Mocker is used to replace `time.time` with a mock that always returns a predictable value, and then we use several assertions to confirm that the attributes of the controller have been initialized to the expected values.

```
    def test_with_when(self):
        controller = pid.PID(P=0.5, I=0.5, D=0.5,
                             setpoint=1, initial=12,
                             when=43)
        self.assertEqual(controller.gains, (0.5, 0.5, 0.5))
        self.assertAlmostEqual(controller.setpoint[0], 1.0)
        self.assertEqual(len(controller.setpoint), 1)
```

```
            self.assertAlmostEqual(controller.previous_time, 43.0)
            self.assertAlmostEqual(controller.previous_error, -11.0)
            self.assertAlmostEqual(controller.integrated_error, 0)
```

This test confirms that the PID constructor works correctly when the `when` parameter is supplied. Unlike the previous test, there's no need to use Mocker, because the outcome of the test is not supposed to be dependant on anything except the parameter values—the current time is irrelevant.

```
    class test_calculate_response(TestCase):
        def test_without_when(self):
            mocker = Mocker()
            mock_time = mocker.replace('time.time')
            mock_time()
            mocker.result(1.0)
            mock_time()
            mocker.result(2.0)
            mock_time()
            mocker.result(3.0)
            mock_time()
            mocker.result(4.0)
            mock_time()
            mocker.result(5.0)
            mocker.replay()
            controller = pid.PID(P=0.5, I=0.5, D=0.5,
                                 setpoint=0, initial=12)
            self.assertEqual(controller.calculate_response(6), -3)
            self.assertEqual(controller.calculate_response(3), -4.5)
            self.assertEqual(controller.calculate_response(-1.5), -0.75)
            sel+f.assertEqual(controller.calculate_response(-2.25),
    -1.125)
            mocker.restore()
            mocker.verify()
```

The tests in this class describe the intended behavior of the `calculate_response` method. This first test checks the behavior when the optional `when` parameter is not supplied, and mocks `time.time` to make that behavior predictable.

```
        def test_with_when(self):
            controller = pid.PID(P=0.5, I=0.5, D=0.5,
                                 setpoint=0, initial=12,
                                 when=1)
            self.assertEqual(controller.calculate_response(6, 2), -3)
            self.assertEqual(controller.calculate_response(3, 3), -4.5)
            self.assertEqual(controller.calculate_response(-1.5, 4),
    -0.75)
            self.assertEqual(controller.calculate_response(-2.25, 5),
    -1.125)
```

In this test, the `when` parameter is supplied, so there is no need to mock `time.time`. We just have to check that the result is what we expected.

The actual tests that we performed are the same ones that were written in the doctest. So far, all that we see is a different way of expressing them.

The first thing to notice is that the test file is divided up into classes that inherit from `unittest.TestCase`, each of which contains one or more test methods. The name of each test method begins with the word *test*, which is how unittest recognizes that they are tests.

Each test method embodies a single test of a single unit. This gives us a convenient way to structure our tests, grouping together related tests into the same class, so that they're easier to find.

Putting each test into its own method means that each test executes in an isolated namespace, which makes it somewhat easier to keep unittest-style tests from interfering with each other, relative to doctest-style tests. It also means that unittest knows how many unit tests are in your test file, instead of simply knowing how many expressions there are (you may have noticed that doctest counts each `>>>` line as a separate test). Finally, putting each test in its own method means that each test has a name, which can be a valuable feature.

Tests in unittest don't directly care about anything that isn't part of a call to one of the assert methods of `TestCase`. That means that when we're using Mocker, we don't have to be bothered about the mock objects that get returned from demonstration expressions, unless we want to use them. It also means that we need to remember to write an assert describing every aspect of the test that we want to have checked. We'll go over the various assertion methods of `TestCase` shortly.

Tests aren't of much use, if you can't execute them. For the moment, the way we'll be doing that is by calling `unittest.main` when our test file is executed as a program by the Python interpreter. That's about the simplest way to run unittest code, but it's cumbersome when you have lots of tests spread across lots of files. We'll be learning about tools to address that problem in the next chapter.

`if __name__ == '__main__':` might look strange to you, but its meaning is fairly straight forward. When Python loads any module, it stores that module's name in a variable called `__name__` within the module (unless the module is the one passed to the interpreter on the command line). That module always gets the string `'__main__'` bound to its `__name__` variable. So, `if __name__ == '__main__':` means—if this module was executed directly from the command line.

Assertions

Assertions are the mechanism that we use to tell unittest what the important outcomes of the test are. By using appropriate assertions, we can tell unittest exactly what to expect from each test.

assertTrue

When we call `self.assertTrue(expression)`, we're telling unittest that the expression must be true in order for the test to be a success.

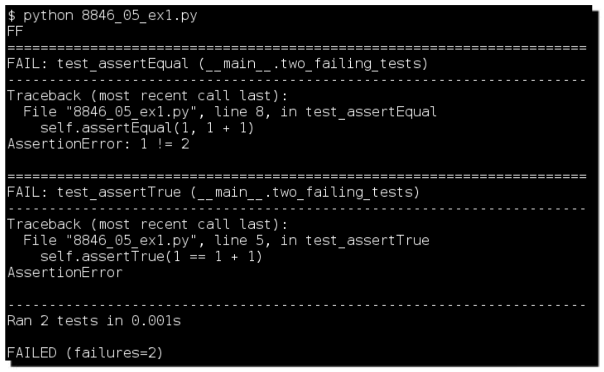
This is a very flexible assertion, since you can check for nearly anything by writing the appropriate boolean expression. It's also one of the last assertions you should consider using, because it doesn't tell unittest anything about the kind of comparison you're making, which means that unittest can't tell you as clearly what's gone wrong if the test fails.

For an example of this, consider the following test code which contains two tests that are guaranteed to fail:

```
from unittest import TestCase, main
class two_failing_tests(TestCase):
    def test_assertTrue(self):
        self.assertTrue(1 == 1 + 1)
    def test_assertEqual(self):
        self.assertEqual(1, 1 + 1)
if __name__ == '__main__':
    main()
```

It might seem like the two tests are interchangeable, since both test the same thing. Certainly they'll both fail (or in the unlikely event that one equals two, they'll both pass), so why prefer one over the other?

Take a look at what happens when we run the tests (and also notice that the tests were not executed in the same order as they were written; tests are totally independent of each other, so that's okay, right?):



```
$ python 8846_05_ex1.py
FF
======================================================================
FAIL: test_assertEqual (__main__.two_failing_tests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "8846_05_ex1.py", line 8, in test_assertEqual
    self.assertEqual(1, 1 + 1)
AssertionError: 1 != 2

======================================================================
FAIL: test_assertTrue (__main__.two_failing_tests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "8846_05_ex1.py", line 5, in test_assertTrue
    self.assertTrue(1 == 1 + 1)
AssertionError

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=2)
```

Do you see the difference? The `assertTrue` test was able to correctly determine that the test should fail, but it didn't know enough to report any useful information about why it failed. The `assertEqual` test, on the other hand, knew first of all that it was checking that two expressions were equal, and second it knew how to present the results, so that they would be most useful: by evaluating each of the expressions that it was comparing and placing a `!=` symbol between the results. It tells us both what expectation failed, and what the relevant expressions evaluate to.

assertFalse

The `assertFalse` method will succeed when the `assertTrue` method would fail, and vice versa. It has the same limits in terms of producing useful output that `assertTrue` has, and the same flexibility in terms of being able to test nearly any condition.

assertEqual

As mentioned in the `assertTrue` discussion, the `assertEqual` assertion checks that its two parameters are in fact equal, and reports a failure if they are not, along with the actual values of the parameters.

assertNotEqual

The `assertNotEqual` assertion fails whenever the `assertEqual` assertion would have succeeded, and vice versa. When it reports a failure, its output indicates that the values of the two expressions are equal, and provides you with those values.

assertAlmostEqual

As we've seen before, comparing floating point numbers can be troublesome. In particular, checking that two floating point numbers are equal is problematic, because things that you might expect to be equal—things that, mathematically, are equal—may still end up differing down among the least significant bits. Floating point numbers only compare equal when every bit is the same.

To address that problem, unittest provides `assertAlmostEqual`, which checks that two floating point values are almost the same; a small amount of difference between them is tolerated.

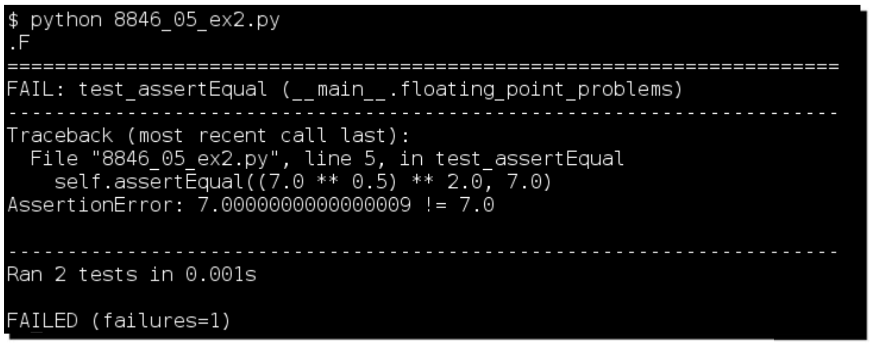
Lets look at this problem in action. If you take the square root of 7, and then square it, the result should be `7`. Here's a pair of tests that check that fact:

```
from unittest import TestCase, main
class floating_point_problems(TestCase):
    def test_assertEqual(self):
        self.assertEqual((7.0 ** 0.5) ** 2.0, 7.0)
    def test_assertAlmostEqual(self):
        self.assertAlmostEqual((7.0 ** 0.5) ** 2.0, 7.0)
if __name__ == '__main__':
    main()
```

The `test_assertEqual` method checks that $\overset{-2}{\boxed{7}} = 7^{\frac{1}{2}^{2}} = 7$, which is true in reality. In the more specialized number system available to computers, though, taking the square root of 7 and then squaring it doesn't quite get us back to 7, so this test will fail. More on that in a moment.

Test `test_assertAlmostEqual` method checks that $\overset{-2}{\boxed{7}} = 7^{\frac{1}{2}^{2}} \approx 7$, which even the computer will agree is true, so this test should pass.

Running those tests produces the following, although the specific number that you get back instead of `7` may vary depending on the details of the computer the tests are being run on:



```
$ python 8846_05_ex2.py
.F
======================================================================
FAIL: test_assertEqual (__main__.floating_point_problems)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "8846_05_ex2.py", line 5, in test_assertEqual
    self.assertEqual((7.0 ** 0.5) ** 2.0, 7.0)
AssertionError: 7.0000000000000009 != 7.0

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

Unfortunately, floating point numbers are not precise, because the majority of numbers on the real number line can not be represented with a finite, non-repeating sequence of digits, much less a mere 64 bits. Consequently, what you get back from evaluating the mathematical expression is not quite `7`. It's close enough for government work though—or practically any other sort of work as well—so we don't want our test to quibble over that tiny difference. Because of that, we should use `assertAlmostEqual` and `assertNotAlmostEqual` when we're comparing floating point numbers for equality.

> This problem doesn't generally carry over into other comparison operators. Checking that one floating point number is less than the other, for example, is very unlikely to produce the wrong result due to insignificant errors. It's only in cases of equality that this problem bites us.

assertNotAlmostEqual

The `assertNotAlmostEqual` assertion fails whenever the `assertAlmostEqual` assertion would have succeeded, and vice versa. When it reports a failure, its output indicates that the values of the two expressions are nearly equal, and provides you with those values.

assertRaises

As always, we need to make sure that our units correctly signal errors. Doing the right thing when they receive good inputs is only half the job; they need to do something reasonable when they receive bad inputs, as well.

The `assertRaises` method checks that a callable (a callable is a function, a method, or a class. A callable can also be an object of any arbitrary type, so long as it has a `__call__` method) raises a specified exception, when passed a specified set of parameters.

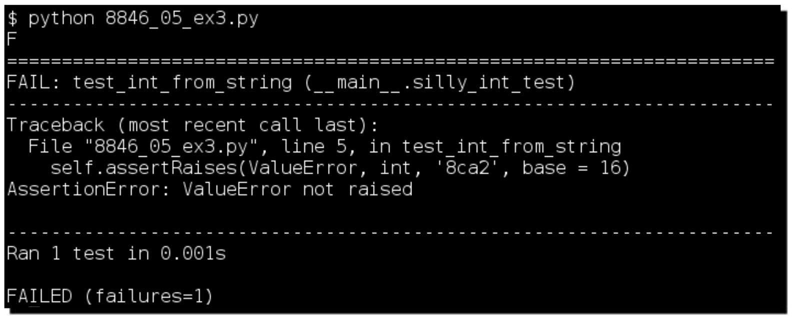
This assertion only works with callables, which means that you don't have a way of checking that other sorts of expressions raise an expected exception. If that doesn't fit the needs of your test, it's possible to construct your own test using the `fail` method, described below.

To use `assertRaises`, first pass it the expected exception, then pass the callable, and then the parameters that should be passed to the callable when it's invoked.

Here's an example test using `assertRaises`. This test ought to fail, because the callable won't raise the expected exception. `'8ca2'` is perfectly acceptable input to `int`, when you're also passing it `base = 16`. Notice that `assertRaises` will accept any number of positional or keyword arguments, and pass them on to the callable on invocation.

```
from unittest import TestCase, main

class silly_int_test(TestCase):
    def test_int_from_string(self):
        self.assertRaises(ValueError, int, '8ca2', base = 16)

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

When we run that test, it fails (as we knew it would) because `int` didn't raise the exception we told `assertRaises` to expect.



```
$ python 8846_05_ex3.py
F
======================================================================
FAIL: test_int_from_string (__main__.silly_int_test)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "8846_05_ex3.py", line 5, in test_int_from_string
    self.assertRaises(ValueError, int, '8ca2', base = 16)
AssertionError: ValueError not raised

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

FAILED (failures=1)
```

If an exception is raised, but it's not the one you told unittest to expect, unittest considers that an error. An error is different from a failure. A failure means that one of your tests has detected a problem in the unit it's testing. An error means that there's a problem with the test itself.

fail

When all else fails, you can fall back on `fail`. When the code in your test calls `fail`, the test fails.

What good does that do? When none of the assert methods does what you need, you can instead write your checks in such a way that `fail` will be called if the test does not pass. This allows you to use the full expressiveness of Python to describe checks for your expectations.

Let's take a look at an example. This time, we're going to test on a less-than operation, which isn't one of the operations directly supported by an assert method. Using `fail`, it's easy to implement the test anyhow.

```
from unittest import TestCase, main

class test_with_fail(TestCase):
    def test_less_than(self):
        if not (2.3 < 5.6):
            self.fail('2.3 is not less than 5.6, but it should be')

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

A couple of things to notice here: first of all, take note of the `not` in the `if` statement. Since we want to run `fail` if the test should not pass, but we're used to describing the circumstances when the test should succeed, a good way to write the test is to write the success condition, and then invert it with `not`. That way we can continue thinking in the way we're used to when we use fail. The second thing to note is that you can pass a message to fail when you call it, which will be printed out in unittest's report of failed tests. If you choose your message carefully, it can be a big help.

There's no screen capture of what to expect from running this test, because the test should pass, and the report wouldn't contain anything interesting. You might experiment with changing the test around and running it, to see what happens.

Pop quiz – basic unittest knowledge

1. What is the unittest equivalent of this doctest?

```
>>> try:
...     int('123')
... except ValueError:
...     pass
... else:
...     print 'Expected exception was not raised'
```

2. How do you check whether two floating point numbers are equal?
3. When would you choose to use `assertTrue`? How about `fail`?

Have a go hero – translating into unittest

Look back at some of the tests we write in the previous chapters, and translate them from doctests into unittests. Given what you already know of unittest, you should be able to translate any of the tests.

While you're doing this, think about the relative merits of unittest and doctest for each of the tests you translate. The two systems have different strengths, so it makes sense that each will be the more appropriate choice for different situations. When is doctest the better choice, and when is unittest?

Test fixtures

Unittest has an important and highly useful capability that doctest lacks. You can tell unittest how to create a standardized environment for your unit tests to run inside, and how to clean up that environment when it's done. This ability to create and destroy a standardized test environment is a test fixture. While test fixtures doesn't actually make any tests possible that were impossible before, they can certainly make them shorter and less repetitive.

Time for action – testing database-backed units

Many programs need to access a database for their operation, which means that many of the units those programs are made of also access a database. The point is that the purpose of a database is to store information and make it accessible in arbitrary other places (in other words, databases exist to break the isolation of units). (The same problem applies to other information stores as well: for example, files in permanent storage.)

How do we deal with that? After all, just leaving the units that interact with the database untested is no solution. We need to create an environment where the database connection works as usual, but where any changes that are made do not last. There are a few different ways we could do that, but no matter what the details are, we need to set up the special database connection before each test that uses it, and we need to destroy any changes after each such test.

Unittest helps us to do that by providing test fixtures via the `setUp` and `tearDown` methods of the `TestCase` class. These methods exist for us to override, with the default versions doing nothing.

Here's some database-using code (let's say it exists in a file called `employees.py`), for which we'll write tests:

This code uses the `sqlite3` database which ships with Python. Since the `sqlite3` interface is compatible with Python's DB-API 2.0, any database backend that you find yourself using will have a similar interface to what you see here.

```
class employees:
    def __init__(self, connection):
        self.connection = connection

    def add_employee(self, first, last, date_of_employment):
        cursor = self.connection.cursor()
        cursor.execute('''insert into employees
                            (first, last, date_of_employment)
                          values
                            (:first, :last, :date_of_
employment)''',
                       locals())
        self.connection.commit()

        return cursor.lastrowid

    def find_employees_by_name(self, first, last):
        cursor = self.connection.cursor()
        cursor.execute('''select * from employees
                          where
                            first like :first
```

```
                                   and
                                     last like :last''',
                                locals())
                for row in cursor:
                    yield row
            def find_employees_by_date(self, date):
                cursor = self.connection.cursor()
                cursor.execute('''select * from employees
                                  where date_of_employment = :date''',
                               locals())
                for row in cursor:
                    yield row
```

1. We'll start writing the tests by importing the modules that we need and introducing our `TestCase` class.

```
from unittest import TestCase, main
from sqlite3 import connect, PARSE_DECLTYPES
from datetime import date
from employees import employees

class test_employees(TestCase):
```

2. We need a `setUp` method to create the environment that our tests depend on. In this case, that means creating a new database connection to an in-memory-only database, and populating that database with the needed tables and rows.

```
    def setUp(self):
        connection = connect(':memory:',
                             detect_types=PARSE_DECLTYPES)
        cursor = connection.cursor()
        cursor.execute('''create table employees
                            (first text,
                             last text,
                             date_of_employment date)''')
        cursor.execute('''insert into employees
                            (first, last, date_of_employment)
                          values
                            ("Test1", "Employee", :date)''',
                       {'date': date(year = 2003,
                                     month = 7,
                                     day = 12)})
        cursor.execute('''insert into employees
                            (first, last, date_of_employment)
                          values
                            ("Test2", "Employee", :date)''',
                       {'date': date(year = 2001,
                                     month = 3,
                                     day = 18)})
        self.connection = connection
```

3. We need a `tearDown` method to undo whatever the `setUp` method did, so that each test can run in an untouched version of the environment. Since the database is only in memory, all we have to do is close the connection, and it goes away. `tearDown` may end up being much more complicated in other scenarios.

```
    def tearDown(self):
        self.connection.close()
```

4. Finally, we need the tests themselves, and the code to execute the tests.

```
    def test_add_employee(self):
        to_test = employees(self.connection)
        to_test.add_employee('Test1', 'Employee', date.today())
        cursor = self.connection.cursor()
        cursor.execute('''select * from employees
                          order by date_of_employment''')
        self.assertEqual(tuple(cursor),
                         (('Test2', 'Employee', date(year=2001,
                                                      month=3,
                                                      day=18)),
                          ('Test1', 'Employee', date(year=2003,
                                                      month=7,
                                                      day=12)),
                          ('Test1', 'Employee', date.today())))
    def test_find_employees_by_name(self):
        to_test = employees(self.connection)
        found = tuple(to_test.find_employees_by_name('Test1',
'Employee'))
        expected = (('Test1', 'Employee', date(year=2003,
                                               month=7,
                                               day=12)),)
        self.assertEqual(found, expected)
    def test_find_employee_by_date(self):
        to_test = employees(self.connection)
        target = date(year=2001, month=3, day=18)
        found = tuple(to_test.find_employees_by_date(target))
        expected = (('Test2', 'Employee', target),)
        self.assertEqual(found, expected)
if __name__ == '__main__':
    main()
```

What just happened?

We used a `setUp` method for our `TestCase`, along with a matching `tearDown` method. Between them, these methods made sure that the environment in which the tests were executed was the one they needed (that was `setUp`'s job) and that the environment of each test was cleaned up after the test was run, so that the tests didn't interfere with each other (which was the job of `tearDown`). Unittest made sure that `setUp` was run once before each test method, and that `tearDown` was run once after each test method.

Because a test fixture—as defined by `setUp` and `tearDown`—gets wrapped around every test in a `TestCase` class, the `setUp` and `tearDown` for `TestCase` classes that contain too many tests can get very complicated and waste a lot of time dealing with details that are unnecessary for some of the tests. You can avoid that problem by simply grouping together, those tests that require specific aspects of the environment into their own `TestCase` classes. Give each `TestCase` an appropriate `setUp` and `tearDown`, only dealing with those aspects of the environment that are necessary for the tests it contains. You can have as many `TestCase` classes as you want, so there's no need to skimp on them when you're deciding which tests to group together.

Notice how simple the `tearDown` method that we used was. That's usually a good sign: when the changes that need to be undone in the `tearDown` method are simple to describe, it often means that you can be sure of doing it perfectly. Since any imperfection of the `tearDown` makes it possible for tests to leave behind stray data that might alter how other tests behave, getting it right is important. In this case, all of our changes were confined to the database, so getting rid of the database does the trick.

Pop quiz – test fixtures

1. What is the purpose of a test fixture?
2. How is a test fixture created?.
3. Can a test fixture have a `tearDown` method without a `setUp`? How about `setUp` without `tearDown`?

Have a go hero – file path abstraction

Below is a class definition that describes an abstraction of file paths. Your challenge is to write unit tests (using unittest) that check each of the methods of the class, making sure that they behave as advertised. You will need to use a test fixture to create and destroy a sandbox area in the filesystem for your tests to operate on.

Because doctest doesn't support test fixtures, writing these tests using that framework would be quite annoying. You'd have to duplicate the code to create the environment before each test, and the code to clean it up after each test. By using `unittest`, we can avoid that duplication.

There are several things about this class that are wrong, or at least not as right as they ought to be. See if you can catch them with your tests.

```
from os.path import isfile, isdir, exists, join
from os import makedirs, rmdir, unlink

class path:
    r"""
    Instances of this class represent a file path, and facilitate
    several operations on files and directories.

    Its most surprising feature is that it overloads the division
    operator, so that the result of placing a / operator between two
    paths (or between a path and a string) results in a longer path,
    representing the two operands joined by the system's path
    separator character.
    """

    def __init__(self, target):
        self.target = target

    def exists(self):
        return exists(self.target)

    def isfile(self):
        return isfile(self.target)

    def isdir(self):
        return isdir(self.target)

    def mkdir(self, mode = 493):
        makedirs(self.target, mode)

    def rmdir(self):
        if self.isdir():
            rmdir(self.target)
        else:
            raise ValueError('Path does not represent a directory')

    def delete(self):
        if self.exists():
            unlink(self.target)
        else:
            raise ValueError('Path does not represent a file')

    def open(self, mode = "r"):
        return open(self.target, mode)

    def __div__(self, other):
        if isinstance(other, path):
            return path(join(self.target, other.target))
        return path(join(self.target, other))

    def __repr__(self):
        return '<path %s>' % self.target
```

Integrating with Python Mocker

You've used Mocker enough to see the repetitiveness involved in creating a mocking context at the beginning of the text and calling its `verify` and `restore` methods at the end. Mocker simplifies this for you by providing a class called `MockerTestCase` in the mocker module. `MockerTestCase` behaves just like a normal unittest `TestCase`, except that for each test, it automatically creates a mocking context, which it then verifies and restores after the test. The mocking context is stored in `self.mocker`.

The following example demonstrates `MockerTestCase` by using it to write a test involving a mock of `time.time`. Before the test gets executed, a mocking context is stored in `self.mocker`. After the test is run, the context is automatically verified and restored.

```
from unittest import main
from mocker import MockerTestCase
from time import time
class test_mocker_integration(MockerTestCase):
    def test_mocking_context(self):
        mocker = self.mocker
        time_mock = mocker.replace('time.time')
        time_mock()
        mocker.result(1.0)
        mocker.replay()
        self.assertAlmostEqual(time(), 1.0)
if __name__ == '__main__':
    main()
```

The above is a simple test that checks that the current time is `1.0`, which it would not be if we didn't mock `time.time`. Instead of creating a new Mocker instance, we have one already available to us as `self.mocker`, so we use that. We also get to leave off the calls to `verify` and `restore`, because the `MockerTestCase` takes care of that for us.

Summary

This chapter contained a lot of information about how to use the unittest framework to write your tests.

Specifically, we covered how to use unittest to express concepts that you were already familiar with from doctest, differences and similarities between unittest and doctest, how to use test fixtures to embed your tests in a controlled and temporary environment, and how to use Python Mocker's `MockerTestCase` to simplify the integration of unittest and Mocker.

Until now, we've been running tests individually, or in small groups, by directly instructing Python to run them. Now that we've learned about unittest, we're ready to talk about managing and executing large bodies of tests, which is the topic of the next chapter.