
Chapter 2
Writing Unit Tests

AN APPLICATION IS one of the great examples of the whole being greater than the sum of its parts. As a whole, the application potentially delivers a huge amount of value. Its component parts perform specific functions, which can be useful on their own but deliver more when combined. That said, it is clear that if any of the smaller parts of the application do not function as expected, the application as a whole is flawed or fails completely. This is where unit testing comes in.

In this chapter you learn all about the fundamental testing practice of unit tests. I will cover exactly what a unit test is, how to write one and provide you examples that assist your learning. There is also a comprehensive breakdown of some of the most widely used unit test methods, which you can refer to as a guide when writing any unit tests yourself. This chapter also seeks to convey some of the standards that Python developers expect when they read over unit tests and shows you how you can put them into practice so they become second nature. Finally, this chapter rounds off with some more advanced techniques in unit testing to give you an insight into the further usages and implementations of unit tests.

What Is Unit Testing?

In unit testing, you look to cover the application's functionality at its most basic level. Test each individual unit of code, typically a method, in isolation to see if given certain conditions it responds in the expected way (see Figure 2-1). Breaking testing down to this level gives you confidence that each part of the application will behave as expected and enables you to cover edge cases where the unexpected happens and deal with them accordingly.

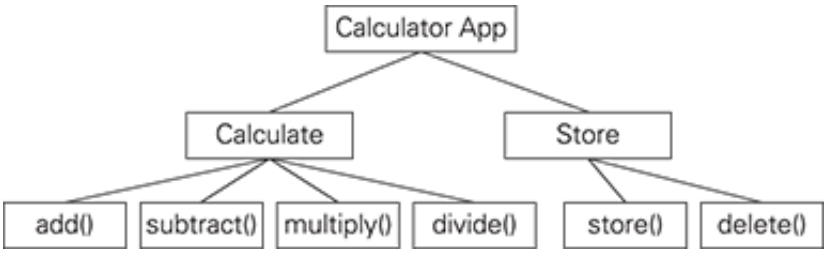

Figure 2-1: Example application structure showing the classes and methods. The methods are the "units" you will test.

In the preceding example, the methods highlighted are the individual units of this application that you need to test. If you know that each of the `calculate` class's methods work as expected, you can be confident that the calculate feature of your application has been delivered to your expectations.

For instance, you may wish to test whether the result of calling the method with two numbers actually adds them together to produce the correct sum. Breaking your code down into these units makes the testing process easier. When dealing with a small unit of an application, you have a clear understanding of its responsibilities and the things that can go wrong with the specific piece of code, thus enabling you to cover the unit with the appropriate tests.

Furthermore, when testing in this manner it usually becomes obvious if you have broken down the code into a good-sized unit. If you have to write many different tests to cover all the different possibilities that the method can go through, your method may be too large and you should consider refactoring it into two or more methods with fine-grained responsibilities. Conversely, there may be cases where your method is too simple and could be combined with some other functionality to create a more useful method. As a programmer with experience, you should start to get a feel for a good-sized method. Ten lines is often a good rule of thumb to follow. There are, of course, plenty of cases where you need more or less than this arbitrary number of lines, and as a programmer your common sense should guide you to provide the most readable code.

The tests that you write are a story that explains your code. What would you want to read or see when first reading through the code and trying to understand what it does? Clear, concise naming conventions of variables, class names, filenames, and tests can all help to make your code clear and easy to maintain for other people.

Testing and, in particular, test driven development (TDD; see Chapter 5), can really help to achieve these goals. TDD forces you to think about your code, and in this moment of careful consideration you can take into account the needs of the application, the design of the code, and how other programmers will interpret your intentions. Use testing to your advantage to make your code cleaner and more efficient. With a good test suite in place, refactoring is easy because you know when you change your code you haven't broken any previous behavior. The tests take the guesswork out of your development process and you can deliver great applications, knowing you have delivered something robust and reliable.

What Should You Test?

A question that many developers ask especially when first starting out is, what should I test? This is an interesting question and also a fair one, as the applications that are being built now are often vast with many complexities. However, unit testing makes the task easier as the whole idea is to focus on the smallest units of code rather than thinking about how to test the large application you are putting together as a whole. Before you write any code you give thought to the kind of tests you will be writing to check the methods will work as expected. As you write more and more unit tests you will gain experience in what works well and what perhaps causes you issues later down the line. For example, a frequent mistake of inexperienced developers is writing very brittle test suites. This means that as code evolves the tests break for reasons other than the functionality changing. The tests are often checking elements of the code or data to too fine a granularity, meaning that as data changes (and not your functionality, which is what you are really testing), the tests fail and you need to go and fix it. Making your tests as flexible as possible while still testing your functionality is the best way to defend against this brittleness.

Another reason to test comes from the process of finding bugs whether in a production environment, the test environment, or while testing your application locally. Whenever you find a bug that affects your application that requires a code change to fix, you should write a test to cover that scenario. By doing this, you ensure that you have covered the defect that caused the problem and with test in place the bug should not reoccur in the future. By adding this layer of defense every time you find a bug (hopefully, not very often) you ensure that your code is more robust in the future as more functionality is added.

Writing Your First Unit Test

By now, you are probably eager to start writing your first unit test. Perhaps you have written tests before but are looking for a refresher in how to write good, concise unit tests. Whatever your Python or testing background, let's start at the beginning with some simple tests for a straightforward application. The examples first show you how to structure your test into a class with the correct naming conventions. Further on in the chapter, you are simply shown snippets of test methods, which you are expected to use with a proper test class.

One of the classic examples for demonstrating unit testing is a small calculator program. Python actually includes a lot of basic math functionality in the standard library. But what if you wrapped that into a simple-to-use command line application that could perform some simple calculations? This first scenario demonstrates how to implement the calculate class of the application example from earlier. Start with the `add` method, which looks something like this.

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

if __name__ == '__main__':
    calc = Calculate()
    result = calc.add(2, 2)
    print result
```

Clearly, this is a very simple class that is just making use of Python's built-in math function for adding and making it into a method you can call in your code. Save this code to a file named `calculate.py`, then execute this and see the result, like so.

```
$ python calculate.py
4
```

Checking Values with the assertEquals Method

You have some working code, so why not write the tests to prove it and look into what could go wrong if the code is used in ways you didn't foresee? Try writing a test that checks to see that if you pass in the two numbers as 2, then you get the result as 4. Using the standard Python library, you can import the `unittest` package. This provides useful methods to make different kinds of assertions (for instance, checking whether something meets some condition) on your method. One of those assertions you can use is the `assertEqual` method. This method allows you to pass in two values and check whether they are equal.

Create a test file called `calculate_test.py`, following the standard naming conventions of using the class name under test and appending with `_test`.

```
import unittest
from app.calculate import Calculate


class TestCalculate(unittest.TestCase):
    def setUp(self):
        self.calc = Calculate()

    def test_add_method_returns_correct_result(self):
        self.assertEqual(4, self.calc.add(2,2))


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

A line-by-line examination shows that this example first imports the functionality you need from Python's `unittest` module. You are also importing your own class, `Calculate`, so that you can test its methods. You do this in the `setUp` method, which is executed before each test, so that you need define your instance only once and have it created before each test. Then you can write your test and again the standard is to append your test name with `append_test` and explain what the test is doing briefly in the rest of the name. Here you are checking if the `add` method returns the correct result. To do this, you make use of the `assertEqual` method provided by the imported `unittest` module. This checks if the first argument is equal to the second. In this example, you are checking whether 4 is equal to the result of calling your `add` method on 2 and 2. In this case, the test passes as your code works and displays the following result in the terminal.

```
$ python test/calculate_test.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s
OK
```

This shows you that your test ran okay, and you ran only the one test. What if things go wrong? You can simulate that by breaking the test. Change `test_add_method_returns_correct_result` to assert that 2 add 3 equals 4, then you should see a failing test.

```
import unittest
from app.calculate import Calculate


class TestCalculate(unittest.TestCase):
    ...
    def test_add_method_returns_correct_result(self):
        self.assertEqual(4, self.calc.add(2,3))
    ...
```

```
$ python test/calculate_test.py
F
======================================================================
FAIL: test_add_method_returns_correct_result (__main__.TestCalculate)
```

```
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/user/workspace/python_testing/test/calculate_test.py", line 12, in test_add_method_returns_correct_result
    self.assertEqual(4, self.calc.add(2,3))
AssertionError: 4 != 5
----------------------------------------------------------------------
Ran 1 test in 0.000s
FAILED (failures=1)
```

You can see that the `unittest` module provides great feedback as to what went wrong and where. You can easily go back and fix the test. Of course, a real failure would usually be caused by your code not meeting your expectation and producing the wrong result. In that case, you would need to go in and fix your code. There you have it. You've written your first, simple unit test.

Checking Exception Handling with assertRaises

You have considered what happens in the normal use cases of your method and what happens if it returns the wrong answer. What about if you passed in something the method doesn't expect? You usually would find in such cases an exception to be raised. The `assertRaises` method, also found in the `unittest` package, provides you with a means to check that a method raises an exception under certain circumstances.

Take a look at your Calculator example again. The `add` method would seem to work with only numbers. What if you pass in two strings like `"Hello"` and `"World"`? Change the piece of code from calculate_test.py to do that.

```
if __name__ == '__main__':
    calc = Calculate()
    result = calc.add("Hello", "World")
    print result
```

Execute the code and see what you get.

```
$ python calculator.py
HelloWorld
```

Because Python isn't strongly typed, you can pass any type of object to `app/calculate.py`'s `add` method and it will try to combine the objects if it can. Again, if this is functionality you would like in your application, you can write a test for it, as before. If you specifically want to stop this behavior, you could try to check the type and allow only numbers to be added.

```
def add(self, x, y):
    if type(x) == int and type(y) == int:
        return x + y
    else:
        raise TypeError("Invalid type: {} and {}".format(type(x),
            type(y)))
```

This works fine when passing in two integers, but what if you try to add a number and a string? You will likely run into problems there.

```
class TestCalculate(unittest.TestCase):

    def setUp(self):
        self.calc = Calculate()

    def test_add_method_returns_correct_result(self):
        self.assertEqual("HelloWorld", self.calc.add("Hello",
"World"))


$ python test/calculate_test.py
E
======================================================================
ERROR: test_add_method_returns_correct_result
 (__main__.TestCalculate)
----------------------------------------------------------------------
------
Traceback (most recent call last):
  File
"/Users/username/workspace/python_testing/test/calculate_test.py"
, line 11, in test_add_method_returns_correct_result
    self.assertEqual(4, self.calc.add("Hello", "World"))
  File "/Users/username/workspace/python_testing/app/calculate.py",
line 10, in add
    raise TypeError("Invalid type: {} and {}".format(type(x),
type(y)))
TypeError: Invalid type: <type 'str'> and <type 'str'>
----------------------------------------------------------------------
Ran 1 test in 0.000s
```

As expected, your code raises the error with the message you have defined to indicate what the problem is. As the types of the object passed in are strings, they don’t meet the if statement’s criteria and so it raises a `TypeError`.

Now that you have defined this as wanted behavior from your method, you can write a test to cover this. Write new test that checks that the `TypeError` is raised when you do pass in strings. For this type of test, you make use of the `assertRaises` method that `unittest` provides. The unit test method `assertRaises` takes three arguments. The first is the type of exception you expect to be raised, in this case `TypeError`. The second is the method under test, in this case `self.calc.add` that you expect to raise this exception. The final value passed in is the argument to the method under test, in this case the string `"Hello"`.

```
def test_add_method_raises_typeerror_if_not_ints(self):
    self.assertRaises(TypeError, self.calc.add, "Hello",
        "World")
$ python test/calculate_test.py
..
----------------------------------------------------------------------
------
Ran 2 tests in 0.000s
OK
```

Following the PEP-8 Standard

As I have been introducing you to unit testing in Python, it should be clear that various patterns and standards are followed within the Python community. Some of them are enforced by tools you may wish to use, such as prepending a test name with `test_` to allow runners such as Nose to find tests to execute. Others are merely standards set by Python developers to keep readability and reuse of code high as it is shared between developers. It helps to give Python code a consistent look and feel that experienced developers are familiar with, and if teams adhere to the accepted standards then when developers move to a new Python project, many aspects of the code should feel familiar.

All Python developers code should conform to the standards outlined within the PEP-8 document (available on the Python website at http://legacy.python.org/dev/peps/pep-0008/). Guido van Rossum, creator of the Python language, along with Barry Warsaw and Nick Coghlan, writes the style guide. The document is one of the most famous PEPs (Python Enhancement Proposals) and also one of the earliest. PEPs are put forward as suggestions for changes to the language or how to use it. PEP-8 focuses on the styling of code and puts forward some of the fundamental principles when writing Python code and tests, such as:

- **Indents:** Four spaces for each indentation

  ```
  def foo():
      print "Hello, World!"
  ```

- **Maximum line length:** 80 characters.
- **Blank lines:** Two between import, class, and function definitions. One between method definitions inside a class.
- **Import statements:** Should be one per line.

  ```
  import os
  import sys
  ```

- **Class names**: Should have capitals for the first letter of each word.

  ```
  class MyClass(object):
  ```

- **Method names:** Should use all lowercase and underscores to separate words.

  ```
  def my_method_example():
  ```

You should endeavor to maintain these standards and use them throughout the code you write. You will also see them followed throughout this book. You can find the whole PEP-8 document and others at www.python.org/dev/peps/. Fortunately, tools have been created to keep your code in check with the standard, such as PyLint. I cover them in detail in Chapter 9.

Unit Test Structure

When structuring your project, you can follow some clear standards to make your application's code more accessible to other Python developers. These simple rules are easy to apply and result in a uniform structure to make it easy to find the test and code files you need.

- Unit tests should be placed under a `test/unit` directory at the top level of your project folder.
- All folders within the application's code should be mirrored by test folders under `test/unit`, which will have the unit tests for each file in them. For example, `app/data` should have a mirrored folder of `test/unit/app/data`.
- All unit test files should mirror the name of the file they are testing, with `_test`as the suffix. For example, `app/data/data_interface.py` should have a test file of `test/unit/app/data/data_interface_test.py`.

To illustrate these rules even more, take a look at one example of how you might structure a Flask project. Flask is the Python web framework package, which you can read more about at http://flask.pocoo.org.

```
example_app/
  example_app/
    __init__.py
    static/
    templates/
    app.py
  test/
     __init__.py
```

```
    unit/
      __init__.py
      app_test.py
```

The `__init__.py` files indicate that the folder is a Python package so that you can import them into other Python files. For example, in `app_test.py`, you need to import methods from `app.py` so that you can test it.

You are, of course, free to structure your project however you like. This is simply a recommended structure that many developers follow.

Additional Unit Test Examples

By now, you have written your first unit test, been introduced to the multitude of testing methods you have at your disposal, and looked at the standards and structures you should follow when creating your unit tests.

Now take a look at some additional examples of unit tests in action and highlight some key points to note against each one.

Getting Clever with assertRaises

In certain situations, `assertRaises` does not appear to work as expected. For instance, when the exception-causing situation is due to calling a method that does not exist on the object, you get an `AttributeError`. However, when you attempt to test for this in the usual manner, you get the following result:

```
def test_assert_raises(self):
    self.assertRaises(AttributeError, [].get)

$ python test/example_test.py
E
======================================================================
ERROR: test_assert_raises (__main__.TestExample)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/username/workspace/python_testing/test/example_test.py", line 77, in test_assert_raises
    self.assertRaises(AttributeError, [].get)
AttributeError: 'list' object has no attribute 'get'

----------------------------------------------------------------------
Ran 19 tests in 0.001s
FAILED (errors=1)
```

Although your expected exception has been raised, the test fails because the exception is not caught by the test method. Fortunately, the `assertRaises` method provides the capability to use it as a context manager. This means you can execute any code you like within the context of `assertRaises`, and if the exception is raised it will be caught and your test will pass as expected.

```
def test_assert_raises(self):
    with self.assertRaises(AttributeError):
        [].get

$ python test/example_test.py
.
----------------------------------------------------------------------
Ran 19 tests in 0.001s

OK
```

Making Your Life Easier with setUp

Unit testing often includes a lot of repeated code. You generally need to create instances of classes to be able to use the methods on them in multiple tests. Following good software development practices such as D.R.Y (**D**on't **R**epeat **Y**ourself) and *Clean Code: A Handbook of Agile Software Craftsmanship* by Robert Cecil Martin (Prentice–Hall, 978-0132350884), you should avoid duplicating code and keep tests as succinct as possible.

Following these principles means that changes to your tests are kept to a minimum; a mistake in duplicated code will need to be changed everywhere it was used. It also ensures your tests are easier to debug. If the test is literally executing the code it is designed to test as opposed to multiple lines of setup, then the developer will be able to clearly see the point of failure and aid in getting the problem fixed.

This is where the `setUp` method comes in. Although oddly named by Python conventions (it perhaps should be `set_up` or `setup`), this aspect of unit testing is powerful and minimizes the code you need to write down. Next I use the calculator example with and without the `setUp` method to illustrate.

Without Setup

```
class TestCalculate(unittest.TestCase):
    def test_add_method_returns_correct_result(self):
      calc = Calculate()
        self.assertEqual(4, calc.add(2,2))

    def test_add_method_raises_typeerror_if_not_ints(self):
        calc = Calculate()
        self.assertRaises(TypeError, calc.add, "Hello", "World")

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

With Setup

```
class TestCalculate(unittest.TestCase):

    def setUp(self):
        self.calc = Calculate()

    def test_add_method_returns_correct_result(self):
        self.assertEqual(4, self.calc.add(2,2))

    def test_add_method_raises_typeerror_if_not_ints(self):
        self.assertRaises(TypeError, self.calc.add, "Hello", "World")

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

Even in this simple test case scenario, the addition of the `setUp` method means you only need to create the instance of `Calculate` once for it to be available to all test cases. Imagine how advantageous it is to be able to create this just once if you needed to test many more scenarios than just these two. Say, for example, you hadn't used the `setUp` but created the `Calculate` class in each test case. Say your class had grown and you now had 15 test cases where this is now created. What if the initializer for `Calculate` changed and you now needed to pass in some new variables to the class? Instead of just one change in the `setUp`, you now need to change 15 lines of code.

It should be noted that even through the use of `setUp`, there is nothing to stop you having some test case which doesn't make use of the objects created in the setup. Perhaps you need to test a slightly different scenario, which requires a different `setUp`. In this case you can just use locally created variables rather than the class level objects the `setUp` method will provide. This is more obvious in cases where you need to mock external libraries or calls. For example, a call to a database might need to be mocked the same way for 90% of your tests, so that would be done in the setup. You may then need to mock it differently to test an error case, which you would do in that test only, ignoring the variables created in the setup.

Useful Methods in Unit Testing

This section provides a quick guide to all the different methods available in the unit test package. For each one, a description of its usage and an example are provided. All methods that take an optional argument, `msg=None`, can be provided a custom message that is displayed on failure.

assertEqual(x, y, msg=None)

This method checks to see whether argument `x` equals argument `y`. Under the covers, this method is performing the check using the == definition for the objects.

```
    def test_assert_equal(self):
        self.assertEqual(1, 1)
```

assertAlmostEqual(x, y, places=None, msg=None, delta=None)

On first glance, this method may seem a little strange but in context becomes useful. The method is basically useful around testing calculations when you want a result to be within a certain amount of places to the expected, or within a certain delta.

```
    def test_assert_almost_equal_delta_0_5(self):
        self.assertAlmostEqual(1, 1.2, delta=0.5)

    def test_assert_almost_equal_places(self):
        self.assertAlmostEqual(1, 1.00001, places=4)
```

assertRaises(exception, method, arguments, msg=None)

Given a method and set of arguments to that method, does it raise the exception? Arguments must match the signature of the method or syntax error is raised. Arguments are passed as comma-separated lists, not as part of the method call, as shown in the following example:

```
    def test_assert_raises(self):
        self.assertRaises(ValueError, int, "a")

    def test_assert_raises_alternative(self):
        with self.assertRaises(AttributeError):
           [].get
```

assertDictContainsSubset(expected, actual, msg=None)

Use this method to check whether `actual` contains `expected`. It's useful for checking that part of a dictionary is present in the result, when you are expecting other things to be there also. For example, a large dictionary may be returned and you need to test that only a couple of entries are present.

```
    def test_assert_dict_contains_subset(self):
        expected = {'a': 'b'}
        actual = {'a': 'b', 'c': 'd', 'e': 'f'}
        self.assertDictContainsSubset(expected, actual)
```

assertDictEqual(d1, d2, msg=None)

This method asserts that two dictionaries contain exactly the same key value pairs. For this test to pass, the two dictionaries must be exactly the same, but not necessarily in the same order.

```
    def test_assert_dict_equal(self):
```

```
        expected = {'a': 'b', 'c': 'd'}
        actual = {'c': 'd', 'a': 'b'}
        self.assertDictEqual(expected, actual)
```

assertTrue(expr, msg=None)

Use this method to check the truth value of an expression or result. This method can be useful and has a few interesting caveats. This is because Python's implicit truth behavior, such as numeric values like 0 and 1 have truth-value of `False` and `True`, respectively. Table 2-1 lists some implied truths along with test examples, but more information can be found in the Python documentation.

Table 2-1 Truth values

Value	Truth
0	False
1	True
-1	True
“”	False
“Hello, World!”	True
None	False

```
    def test_assert_true(self):
        self.assertTrue(1)
        self.assertTrue("Hello, World")
```

assertFalse(expr, msg=None)

This method is the inverse of `assertTrue` and is used for checking whether the expression or result under the test is `False`.

```
    def test_assert_false(self):
        self.assertFalse(0)
        self.assertFalse("")
```

assertGreater(a, b, msg=None)

This method allows you to check whether one value is greater than the other. It is essentially a helper method that wraps up the use of `assertTrue` on the expression `a > b`. It displays a helpful message by default when the value is not greater.

```
    def test_assert_greater(self):
        self.assertGreater(2, 1)
```

assertGreaterEqual(a, b, msg=None)

You use this method to check whether one value is greater than *or equal to* another value. Essentially, this wrapper is asserting `True` on `a >= b`. The assertion also gives a nicer message if the expectation is not met.

```
    def test_assert_greater_equal(self):
        self.assertGreaterEqual(2, 2)
```

assertIn(member, container, msg=None)

With this method, you can check whether a value is in a container (hashable) such as a list or tuple. This method is useful when you don't care what the other values are, you just wish to check that a certain value(s) is in the container.

```
    def test_assert_in(self):
        self.assertIn(1, [1,2,3,4,5])
```

assertIs(expr1, expr2)

Use this method to check that `expr1` and `expr2` are identical. That is to say they are the same object. For example, the python code `[] is []` would return `False`, as the creation of each list is a separate object.

```
    def test_assert_is(self):
        self.assertIs("a", "a")
```

assertIsInstance(obj, class, msg=None)

This method asserts that an object is an instance of a specified class. This is useful for checking that the return type of your method is as expected (for instance, if you wish to check that a value is a type of `int`):

```
    def test_assert_is_instance(self):
        self.assertIsInstance(1, int)
```

assertNotIsInstance(obj, class, msg=None)

This reverse of `assertIsInstance` provides an easy way to assert that the object is not a type of the class.

```
    def test_assert_is_not_instance(self):
        self.assertNotIsInstance(1, str)
```

assertIsNone(obj, msg=None)

Use this to easily check if a value is `None`. This method provides a useful standard message if not `None`.

```
    def test_assert_is_none(self):
        self.assertIsNone(None)
```

assertIsNot(expr1, expr2, msg=None)

Using this method, you can check that `expr1` is not the same as `expr2`. This is to say that `expr1` is not the same object as `expr2`.

```
    def test_assert_is_not(self):
        self.assertIsNot([], [])
```

assertIsNotNone(obj, msg=None)

This method checks that the value provided is not `None`. This is useful for checking that your method returns an actual value, rather than nothing.

```
    def test_assert_is_not_none(self):
        self.assertIsNotNone(1)
```

assertLess(a, b, msg=None)

This method checks that the value `a` is less than the value `b`. This is a wrapper method for `assertTrue` on `a < b`.

```
    def test_assert_less(self):
        self.assertLess(1, 2)
```

assertLessEqual(a, b, msg=None)

This method checks that the value `a` is less than or equal to the value `b`. This is a wrapper method for `assertTrue` on `a <= b`.

```
    def test_assert_less_equal(self):
        self.assertLessEqual(2, 2)
```

assertItemsEqual(a, b, msg=None)

This assertion is perfect for testing whether two lists are equal. Lists are unordered; therefore, `assertEqual` on a list can produce intermittent failing tests as the order of the lists may change when running the tests. This can produce a failing test when in fact the two lists have the same contents and are equal.

```
    def test_assert_items_equal(self):
        self.assertItemsEqual([1,2,3], [3,1,2])
```

assertRaises(excClass, callableObj, *args, **kwargs, msg=None)

This assertion is used to check that under certain conditions exceptions are raised. You pass in the exception you expect, the callable that will raise the exception and any arguments to that callable. In the earlier example, this pops the first item from an empty list and results in an `IndexError`.

```
    def test_assert_raises(self):
        self.assertRaises(IndexError, [].pop, 0)
```

Hopefully, the assertions outlined in this section should be all you need to write any unit tests that exercise your application's functionality. A couple of assertions may be missing from this list. You can find the full `unittest` documentation in the Python documentation at: http://docs.python.org/2/library/unittest.html.

Summary

You got down to business in this chapter. You were introduced to the concept of unit testing, breaking down your application into small bite-sized chunks that could be tested in isolation. You wrote your first unit test! By taking a small example application such as a Calculator, you were able to see how to test the individual responsibilities of the class and methods.

You wrote unit tests that make use of the two main assertions in `assertEqual` and `assertRaises` to check happy and unhappy paths through the code. PEP-8 showed Python developers the standards they should adhere to both in code and in tests. By following the guidelines outlined in the PEP-8 document, you can ensure you will write neat, readable Python code that is easily accessible to other Python developers.

A comprehensive guide to the different assertion methods available when unit testing should prove to be a valuable resource to those starting out or brushing up their skills in unit testing. A clear guide to the method names and arguments to be provided makes it easy to get writing unit tests of your own and also maybe discover some methods you weren't aware of.

Finally, I rounded off the chapter with advice on the standard structure and makeup of a typical Python application and its tests. As with PEP-8, following a standard structure ensures it is easier to work on projects with other Python developers, but it is important to do what is best for you and your project should the need arise. Some more advanced unit test cases and improvements completed the chapter to ensure you have everything you need to write good unit tests that exercise your application efficiently.