Test-Driven Development In Python

Aaron Maxwell

aaron@powerfulpython.com

Contents

1 Automated Testing and TDD **3**

1.1 What is Test-Driven Development? 4

1.2 Unit Tests And Simple Assertions 5

1.3 Fixtures And Common Test Setup 11

1.4 Asserting Exceptions 14

1.5 Using Subtests 15

1.6 Final Thoughts 20

Index **22**

Chapter 1

Automated Testing and TDD

Writing automated tests is one of those things that separates average developers from the best in the world. Master this skill, and you will be able to write *far* more complex and powerful software than you ever could before. It's a superpower, and changes the arc of your career.

There are roughly two kinds of readers for this chapter. Some of you have, so far, little or no experience writing automated tests, in any language. This chapter is primarily written for you. It introduces many fundamental ideas of test automation, explains the problems it is supposed to solve, and teaches how to apply Python's tools for doing so.

Or you might be someone with extensive experience using standard test frameworks in other languages: JUnit in Java, PHPUnit in PHP, and so on. Generally speaking, if you have mastered an xUnit framework in another language, and are fluent in Python, you may be able to start skimming the Python's `unittest` module docs[1] and be productive in minutes. Python's test library, `unittest`, was originally based on JUnit 3, and maps very closely to how most xUnit libraries work.[2]

If you are more experienced, I believe it's worth your time to at least skim the chapter, and perhaps study it thoroughly. In writing, I invested a great deal of effort weaving in useful, real-world wisdom - both for software testing in general, and for Python specifically. This includes topics like how to organize Python test code; writing test code which is maintainable; useful, Python-only features like subtests; and even cognitive aspects of programming... getting into an enjoyable, highly productive "flow" state via test-driven development.

[1] https://docs.python.org/3/library/unittest.html

[2] You may be in a third category, having a lot of experience with a non-xUnit testing framework. If so, you should probably pretend you're in the first group. You'll be able to move quickly.

With that in mind, let's start with the core ideas for writing automated tests. We'll then focus on writing them for Python programs.

1.1 What is Test-Driven Development?

An *automated test* is a program that tests another program. Generally, it tests a specific portion of that program: a function, a method, a class, or some group of these things. We call that portion the "system under test". If the system under test is working correctly, the test passes; if not, our test catches that error, and immediately tells us what is wrong. Real applications accumulate many of these tests as development proceeds.

People have different names for different kinds of automated tests: unit tests, integration tests, end-to-end tests, etc. These distinctions can be useful, but we won't need to worry about them right now. They all share the same foundation.

In this chapter, we do *test-driven development*, or TDD. Test-driven development means you start working on each new feature or bugfix by writing the automated test for it **first**. You run that test, verify it fails, and only then do you write the actual code for the feature. You know you are done when the test passes.

This is a different process from implementing the feature first, **then** writing a test for it after. Writing the test first forces you to think through the interfaces of your code, answering the question "how will I know my code is working?" That immediate benefit is useful, but not the whole story.

The greatest mid-term benefits are mostly cognitive. As you become competent and comfortable with test-driven development, you learn to easily get into a state of flow - where you find yourself repeatedly implementing feature after feature, keeping your focus with ease for long periods of time. You can honestly surprise and delight yourself with how much you've accomplished in a few hours of coding.

But the greatest benefits emerge over time. We've all done substantial refactorings of a large code base, changing fundamental aspects of its architecture.[3] Such refactorings - which threaten to break the application in confusing, hidden ways - become straightforward and safe using TDD. You take the existing body of tests, updating where needed and introducing new tests as appropriate. Then all you have to do is make them pass. It may still be a ton of work.

[3] If you haven't done one of these yet, you will.

But you can be fairly confident in the correctness and robustness of the result, instead of hoping and praying.

Among developers who know how to write tests, some love to do test-driven development in their day to day work. Some like to do it part of the time; some hate it, and do it rarely, or never. However, the absolute best way to quickly master unit testing is to strictly do test-driven development for a while. So I'll teach you how to do that. You don't have to do it forever if you don't want to.

Python's standard library ships with two modules for creating unit tests: `doctest` and `unittest`. Most engineering teams prefer `unittest`, as it is more full-featured than `doctest`. This isn't just a convenience. There is a real ceiling of complexity that `doctest` can handle, and real applications will quickly bump up against that limit. With `unittest`, the sky is more or less the limit.

In addition - as noted above - `unittest` maps almost exactly to the xUnit libraries used in many other languages. If you are already familiar with Python, and have used JUnit, PHPUnit, or any other xUnit library in any language, you will feel right at home with unittest. That said, `unittest` has some unique tools and idioms - partly because of differences in the Python language, and partly from unique extensions and improvements. We will learn the best of what `unittest` has to offer as we go along.

1.2 Unit Tests And Simple Assertions

Imagine a class representing an angle:

```
>>> small_angle = Angle(60)
>>> small_angle.degrees
60
>>> small_angle.is_acute()
True
>>> big_angle = Angle(320)
>>> big_angle.is_acute()
False
>>> funny_angle = Angle(1081)
>>> funny_angle.degrees
1
>>> total_angle = small_angle.add(big_angle)
>>> total_angle.degrees
20
```

As you can see, `Angle` keeps track of the angle size, wrapping around so it's in a range of 0 up to 360 degrees. There is also an `is_acute` method, to tell you if its size is under 90 degrees, and an `add` method for arithmetic.[4]

Suppose this `Angle` class is defined in a file named `angle.py`. Here's how we create a simple test for it - in a separate file, named `test_angle.py`:

[4]The object-oriented chapter talks about "magic methods" like `__add__`, which provide a more natural syntax for math-like operations on custom types. This chapter just uses regular methods, in case you haven't read that chapter yet.

```
import unittest
from angle import Angle

class TestAngle(unittest.TestCase):
    def test_degrees(self):
        small_angle = Angle(60)
        self.assertEqual(60, small_angle.degrees)
        self.assertTrue(small_angle.is_acute())
        big_angle = Angle(320)
        self.assertFalse(big_angle.is_acute())
        funny_angle = Angle(1081)
        self.assertEqual(1, funny_angle.degrees)

    def test_arithmetic(self):
        small_angle = Angle(60)
        big_angle = Angle(320)
        total_angle = small_angle.add(big_angle)
        self.assertEqual(20, total_angle.degrees,
                         'Adding angles with wrap-around')
```

As you look over this code, notice a few things:

- There's a class called `TestAngle`. You just define it, not create any instance of it. This subclasses `TestCase`.
- You define two methods, `test_degrees` and `test_arithmetic`.
- Both `test_degrees` and `test_arithmetic` have assertions, using some methods of `TestCase`: `assertEqual`, `assertTrue`, and `assertFalse`.
- The last assertion includes a custom message, as its third argument.

To see how this works, let's define a stub for the `Angle` class in `angles.py`:

```
# angle.py, version 1
class Angle:
    def __init__(self, degrees):
        self.degrees = 0
    def is_acute(self):
        return False
    def add(self, other_angle):
        return Angle(0)
```

This `Angle` class defines all the attributes and methods it is expected to have, but otherwise can't do anything useful. We need a stub like this to verify the test can run correctly, and alert us to the fact that the code isn't working yet.

The `unittest` module is not just used to define tests, but also to run them. You do so on the command line like this:

```
python3 -m unittest test_angles.py
```

When you run the test,[5] and you'll see the following output:

[5] Python 2 requires you to drop the test file's `.py` extension - in other words, passing the test module name. So you invoke it like `python2.7 -m unittest test_angles`. Python 3 lets you do either; we'll always use the test filename in this chapter, but you can use whichever you prefer.

```
$ python3 -m unittest test_angle.py
FF
======================================================
FAIL: test_arithmetic (test_angle.TestAngle)
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_angle.py", line 18, in test_arithmetic
    self.assertEqual(20, total_angle.degrees, 'Adding angles with
       wrap-around')
AssertionError: 20 != 0 : Adding angles with wrap-around

======================================================
FAIL: test_degrees (test_angle.TestAngle)
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_angle.py", line 7, in test_degrees
    self.assertEqual(60, small_angle.degrees)
AssertionError: 60 != 0

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

FAILED (failures=2)
```

Notice:

- Both test methods are shown. They both have a failed assertion highlighted.
- `test_degrees` makes several assertions, but only the first one has been run - once it fails, the others are not executed.
- For each failing assertion, you are given the line number; the expected and actual values; and its test method.
- The custom message in `test_arithmetic` shows up in the output.

This demonstrates one useful way to organize your test code. In a single test module (`test_angle.py`), you define one or more subclasses of `unittest.TestCase`. Here, I just define `TestAngle`, containing tests for the `Angle` class. Within this, I create several test methods, for testing different aspects of the class. And in each of these test methods, I can have as many assertions as makes sense.

Some of the naming conventions matter. It's traditional to start a test class name with the string `Test`, but that is not required; `unittest` will find all subclasses of `TestCase` automatically. But every method must start with the string "test". If it starts with anything else (even "Test"!), `unittest` will not run its assertions.

Running the test and watching it fail is an important first step. It verifies that the test does, in fact, actually test your code. As you write more and more tests, you'll occasionally create the test; run it, expecting it to fail; and find it unexpectedly passes. That's a bug in your test code! Fortunately you ran the test first, so you caught it right away.

In the test code, we defined `test_degrees` before `test_arithmetic`, but they were actually run in the opposite order. It's important to craft your test methods to be self-contained, and not depend on one being run before the other; the order of execution is essentially random.[6]

At this point, we have a correctly failing test. If I'm using version control and working in a branch, this is a good commit point - check in the test code, because it specifies the correct behavior (even if it's presently failing). The next step is to actually make that test pass. Here's one way to do it:

```
# angle.py, version 2
class Angle:
    def __init__(self, degrees):
        self.degrees = degrees % 360
    def is_acute(self):
        return self.degrees < 90
    def add(self, other_angle):
        return Angle(self.degrees + other_angle.degrees)
```

Now when I run my test again, the test passes:

```
python3 -m unittest test_angle1.py
..
----------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

This becomes your second commit in version control.

[6] If you find yourself wanting to run tests in a certain order, this might be better handled with `setUp` and `tearDown`, explained in the next section.

`assertEqual`, `assertTrue` and `assertFalse` will be the most common assertion methods you'll use, along with `assertNotEqual` (which does the opposite of `assertEqual`). Many others are provided, such as `assertIs`, `assertIsNone`, `assertIn`, and `assertIsInstance` - along with "not" variants (e.g. `assertIsNot`). Each takes a optional final message-string argument, like "Adding angles with wrap-around" in `test_arithmetic` above. If the test fails, this is printed in the output, which can give very helpful advice to whomever is troubleshooting a broken test.[7]

If you try checking that two dictionaries are equal, and they are not, the output is tailored to the data type: highlighting which key is missing, or which value is incorrect, for example. This also happens with lists, tuples, and sets, making troubleshooting much easier. What's actually happening is that `unittest` provides certain type-specialized assertions, like `assertDictEqual`, `assertListEqual`, and more. You almost never need to invoke them directly: if you invoke `assertEqual` with two dictionaries, it automatically dispatches to `assertDictEqual`, and similar for the other types. So you get this usefully detailed error reporting for free.

Notice the `assertEqual` lines take two arguments, and I always wrote the expected, correct value first:

```
small_angle = Angle(60)
self.assertEqual(60, small_angle.degrees)
```

It does not matter whether the expected value is first, or second. But it's smart to pick an order and stick with it - at least throughout a single codebase, and maybe for all code you write. Sticking with a consistent order greatly improves the readability of your test output, because you never have to decipher which is which. Believe me, this will save you a lot of time in the long run. If you're on a team, negotiate with them to agree on a consistent order.

1.3 Fixtures And Common Test Setup

As an application grows and you write more tests, you will find yourself writing groups of test methods that start or end with the same lines of code. This repeated code - which does some kind of pretest set-up, and/or after-test cleanup - can be consolidated in the special methods `setUp` and `tearDown`. When defined in your `TestCase` subclass, `setUp` is executed just before each test method starts; `tearDown` is run just after. This is repeated for every single test method.

[7] Which could be you, months or years down the road. Be considerate of your future self!

Here's a realistic example of when you might use it. Imagine working on a tool that saves its state between runs in a special file, in JSON format. We'll call this the "state file". On start, it reads the state from the file; on exit, it rewrites it, if there are any changes. A stub of this class might look like

```
# statefile.py
class State:
    def __init__(self, state_file_path):
        # Load the stored state data, and save
        # it in self.data.
        self.data = { }
    def close(self):
        # Handle any changes on application exit.
```

In fleshing out this stub, we want our tests to verify the following:

- If I add a new key-value pair to the state, it is recorded correctly in the state file.
- If I alter the value of an existing key, that updated value is written to the state file.
- If the state is not changed, the state file's content stays the same.

For each test, we want the state file to be in a known starting state. Afterwards, we want to remove that file, so our tests don't leave garbage files on the filesystem. Here's how the `setUp` and `tearDown` methods accomplish this:

```
import os
import unittest
import shutil
import tempfile
from statefile import State

INITIAL_STATE = '{"foo": 42, "bar": 17}'

class TestState(unittest.TestCase):
    def setUp(self):
        self.testdir = tempfile.mkdtemp()
        self.state_file_path = os.path.join(
            self.testdir, 'statefile.json')
        with open(self.state_file_path, 'w') as outfile:
            outfile.write(INITIAL_STATE)
        self.state = State(self.state_file_path)

    def tearDown(self):
        shutil.rmtree(self.testdir)

    def test_change_value(self):
        self.state.data["foo"] = 21
        self.state.close()
        reloaded_statefile = State(self.state_file_path)
        self.assertEqual(21,
            reloaded_statefile.data["foo"])

    def test_remove_value(self):
        del self.state.data["bar"]
        self.state.close()
        reloaded_statefile = State(self.state_file_path)
        self.assertNotIn("bar", reloaded_statefile.data)

    def test_no_change(self):
        self.state.close()
        with open(self.state_file_path) as handle:
            checked_content = handle.read()
        self.assertEqual(checked_content, INITIAL_STATE)
```

In `setUp`, we create a fresh temporary directory, and write the contents of `INITIAL_DATA` inside. Since we know each test will be working with a `State` object based on that initial data, we go ahead and create that object, and save it in `self.state`. Each test can then work with that object, confident it is in the same consistent starting state, regardless of what any other test method does. In effect, `setUp` creates a private sandbox for each test method.

The tests in `TestState` would all work reliably with just `setUp`. But we also want to clean up the temporary files we created; otherwise, they will accumulate over time with repeated test runs. The `tearDown` method will run after each `test_*` method completes, even if some of its assertions fail. This ensures the temp files and directories are all removed completely.

The generic term for this kind of pre-test preparation is called a *test fixture*. A test fixture is whatever needs to be done before a test can properly run. In this case, we set up the text fixture by creating the state file, and the `State` object. A text fixture can be a mock database; a set of files in a known state; some kind of network connection; or even starting a server process. You can do all these with `setUp`.

`tearDown` is for shutting down and cleaning up the text fixture: deleting files, stopping the server process, etc. For some kinds of tests, a tear-down might not be at all optional. If `setUp` starts some kind of server process, for example, and `tearDown` fails to terminate it, then `setUp` may not be able to run for the next test.

The camel-casing matters: people sometimes misspell them as `setup` or `teardown`, then wonder why they don't seem to be invoked. Also, any uncaught exception in either `setUp` or `tearDown` will cause `unittest` to mark the test method as failing (which means it will clearly show up in the test output), then immediately skip to the next test. For errors in `setUp`, this means none of that test's assertions will run (though it's still marked as failing). For `tearDown`, the test is marked as failing, even if all the individual assertions passed.

1.4 Asserting Exceptions

Sometimes your code is supposed to raise an exception, under certain exceptional conditions. If that condition occurs, and your code does *not* raise the correct exception, that's a bug. How do you write test code for this situation?

You can verify that behavior with a special method of `TestCase`, called `assertRaises`. It's used in a `with` statement in your test; the block under the `with` statement is asserted to raise the exception. For example, suppose you are writing a library that translates Roman numerals into integers. You might define a function called `roman2int`:

```
>>> roman2int("XVI")
16
>>> roman2int("II")
2
```

In thinking about the best way to design this function, you decide that passing nonsensical input to `roman2int` should raise a `ValueError`. Here's how you write a test to assert that behavior:

```
import unittest
from roman import roman2int

class TestRoman(unittest.TestCase):
    def test_roman2int_error(self):
        with self.assertRaises(ValueError):
            roman2int("This is not a valid roman numeral.")
```

If you run this test, and `roman2int` does NOT raise the error, this is the result:

```
$ python3 -m unittest test_roman2int.py
F
======================================================
FAIL: test_roman2int_error (test_roman2int.TestRoman)
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_roman2int.py", line 7, in test_roman2int_error
    roman2int("This is not a valid roman numeral.")
AssertionError: ValueError not raised

------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

When you fix the bug, and `roman2int` raises `ValueError` like it should, the test passes.

1.5 Using Subtests

Python 3 has a new feature called subtests, allowing you to conveniently iterate through a collection of test inputs. Imagine a function called `numwords`, which counts the number of unique words in a string (ignoring punctuation, spelling and spaces):

```
>>> numwords("Good, good morning. Beautiful morning!")
3
```

Suppose you want to test how `numwords` handles excess whitespace. You can easily imagine a dozen different reasonable inputs that will result in the same return value, and want to verify it can handle them all. You might create something like this:

```
class TestWords(unittest.TestCase):
    def test_whitespace(self):
        self.assertEqual(2, numwords("foo bar"))
        self.assertEqual(2, numwords("    foo bar"))
        self.assertEqual(2, numwords("foo\tbar"))
        self.assertEqual(2, numwords("foo   bar"))
        self.assertEqual(2, numwords("foo bar    \t   \t"))
        # And so on, and so on...
```

Seems a bit repetitive, doesn't it? The only thing varying is the argument to `numwords`. We might benefit from using a for loop:

```
    def test_whitespace_forloop(self):
        texts = [
            "foo bar",
            "    foo bar",
            "foo\tbar",
            "foo   bar",
            "foo bar    \t   \t",
            ]
        for text in texts:
            self.assertEqual(2, numwords(text))
```

At first glance, this is certainly more maintainable. If we add new variants, it's just another line in the `texts` list. And if I rename `numwords`, I only need to change it in one place in the test.

However, using a for loop like this creates more problems than it solves. Suppose you run this test, and get the following failure:

```
$ python3 -m unittest test_words_forloop.py
F
======================================================
FAIL: test_whitespace_forloop (test_words_forloop.TestWords)
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_words_forloop.py", line 17, in
     test_whitespace_forloop
    self.assertEqual(2, numwords(text))
AssertionError: 2 != 3

------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

Look closely, and you'll realize that `numwords` returned 3 when it was supposed to return 2. Pop quiz: out of all the inputs in the list, which caused the bad return value?

The way we've written the test, there is no way to know. All you can infer is that at least one of the test inputs produced an incorrect value. You don't know which one. That's the first problem. The second is that the test stops when the first failure happens. If several test inputs are causing errors, it would be helpful to know that right away. (Of course, the original version has this shortcoming too.) Knowing all the failing inputs, and the incorrect results they create, would be *very* helpful for quickly understanding what's going on.

Python 3.4 introduced a new feature, called *subtests*, that gives you the best of all worlds. Our for-loop solution is actually quite close. All we have to do is add one line - can you spot it below?

```
    def test_whitespace_subtest(self):
        texts = [
            "foo bar",
            "    foo bar",
            "foo\tbar",
            "foo   bar",
            "foo bar    \t   \t",
            ]
        for text in texts:
            with self.subTest(text=text):
                self.assertEqual(2, numwords(text))
```

Just inside the for loop, we write `with self.subTest(text=text)`. This creates a context in which assertions can be made, and even fail. Regardless of whether they pass or not, the test continues with the next iteration of the for loop. At the end, *all* failures are collected and reported in the test result output, like this:

```
$ python3 -m unittest test_words_subtest.py

======================================================
FAIL: test_whitespace_subtest (test_words_subtest.TestWords) (text='
   foo\tbar')
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_words_subtest.py", line 16, in
     test_whitespace_subtest
    self.assertEqual(2, numwords(text))
AssertionError: 2 != 3

======================================================
FAIL: test_whitespace_subtest (test_words_subtest.TestWords) (text='
   foo bar    \t   \t')
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_words_subtest.py", line 16, in
     test_whitespace_subtest
    self.assertEqual(2, numwords(text))
AssertionError: 2 != 4

------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=2)
```

Behold the opulence of information in this output:

- Each individual failing input has its own detailed summary.
- We are told what the full value of `text` was.
- We are told what the actual returned value was, clearly compared to the expected value.
- No values are skipped. We can be confident that these two are the *only* failures.

This is MUCH better. The two offending inputs are `"foo\tbar"` and `"foo bar \t \t"`. These are the only values containing tab characters, so you can quickly realize the nature of the bug: tab characters are being counted as separate words.

Let's look at the three key lines of code again:

```
        for text in texts:
            with self.subTest(text=text):
                self.assertEqual(2, numwords(text))
```

The key-value arguments to `self.subTest` are used in reporting the output. They can be anything that helps you understand exactly what is wrong when a test fails. Often you will want to pass everything that varies from the test cases; here, that's only the string passed to `numwords`.

Be clear that in these three lines, the symbol `text` is used for two different things. The `text` variable in the for loop is the same variable that is passed to `numwords` on the last line. In the call to `subTest`, the left-hand side of `text=text` is actually a parameter that is used in the reporting output if the test fails. For example, suppose we wrote it as `input_text` instead:

```
        for text in texts:
            with self.subTest(input_text=text):
                self.assertEqual(2, numwords(text))
```

Then the failure output might look like:

```
FAIL: test_whitespace_subtest (test_words_subtest.TestWords) (
   input_text='foo\tbar')
```

In other words, the l-value `text` in the `assertEqual` line has nothing to do with the argument to `subTest`. Just remember that the arguments to `subTest` are only used in the error output when something goes wrong, and are otherwise ignored completely.

1.6 Final Thoughts

Let's recap the big ideas. Test-driven development means we create the test first, and whatever stubs we need to make the test run. We then run it, and watch it fail. **This is an important step.** You must run the test and see it fail.

This is important for two reasons. You don't really know if the test is correct until you verify that it **can** fail. As you write automated tests more and more over time, you will probably be surprised at how often you write a test, confident in its correctness, only to discover it passes when it should fail. As far as I can tell, every good, experienced software engineer I know still

commonly experiences this... even after doing TDD for many years! This is why we build the habit of always verifying the test fails first.

The second reason is more subtle. As you gain experience with TDD and become comfortable with it, you will find the cycle of writing tests and making them pass lets you get into a state of flow. This means you are enjoyably productive and focused, in a way that is easy to maintain over time.

Is it important that you strictly follow test-driven development? People have different opinions on this, some of them *very* strong. Personally, I went through a period of almost a year where I followed TDD to the letter, very strictly. As a result, I got *very* good at writing tests, and writing high-quality tests very quickly.

Now that I've developed that level of skill, I prefer instead to follow the 80-20 rule, and sometimes the 70-30 or even 50-50 rule. I have noticed that TDD is most powerful when I have great clarity on the software's design, architecture and APIs; it helps me get into an cognitive state that seems accelerated, so that I can more easily maintain my mental focus, and produce quality code faster.

But I find it very hard to write good tests when I don't yet have that clarity... when I am still thinking through how I will structure the program and organize the code. In fact, I find TDD slows me down in that phase, as any test I write will probably have to be completely rewritten several times, if not deleted, before things stabilize. In these situations, I prefer to get a first version of the code working through manual testing, then write the tests afterwards.

To close with the obvious: Experiment to find what approach works best for you, and not just follow what someone else writes that you "should" do. I encourage you to try TDD for a period of time, because of what it will teach you. But be flexible, and at some point step back and evaluate how you want to integrate it into your daily routine.

Index

A
automated tests, 3

D
doctest, 5

F
flow, 4

I
integration tests, 4

J
JUnit, 3, 5

P
PHPUnit, 3, 5

R
refactoring, 4

S
setUp (in unit tests), 11
subtests, 15

T
TDD, 4
tearDown (in unit tests), 11
test fixtures, 11
test-driven development, 4, 20
TestCase class, 6
tests
    automated tests, 3
    integration tests, 4
    unit tests, 3, 4

U
unit tests, 3, 4
unittest, 5

X
xUnit, 3, 5