Table of Contents

Extending Our Functional Test Using the unittest Module

Let’s adapt our test, which currently checks for the default Django "it worked" page, and check instead for some of the things we want to see on the real front page of our site.

Time to reveal what kind of web app we’re building: a to-do lists site! I know, I know, every other web dev tutorial online is also a to-do lists app, or maybe a blog or a polls app. I’m very much following fashion.

The reason is that a to-do list is a really nice example. At its most basic, it is very simple indeed—just a list of text strings—so it’s easy to get a "minimum viable" list app up and running. But it can be extended in all sorts of ways—different persistence models, adding deadlines, reminders, sharing with other users, and improving the client-side UI. There’s no reason to be limited to just "to-do" lists either; they could be any kind of lists. But the point is that it should enable me to demonstrate all of the main aspects of web programming, and how you apply TDD to them.

Using a Functional Test to Scope Out a Minimum Viable App

Tests that use Selenium let us drive a real web browser, so they really let us see how the application *functions* from the user’s point of view. That’s why they’re called *functional tests*.

This means that an FT can be a sort of specification for your application. It tends to track what you might call a *user story*, and follows how the user might work with a particular feature and how the app should respond to them.[1]

Terminology:
Functional Test == End-to-End Test == Acceptance Test

What I call functional tests, some people prefer to call *end-to-end tests*, or, slightly less commonly, *system tests*.

The main point is that these kinds of tests look at how the whole application functions, from the outside. Another name is *black box test*, or *closed box test*, because the test doesn't know anything about the internals of the system under test.

Others also like the name *acceptance tests* (see [acceptance_tests]). This distinction is less about the level of granularity of the test or the system, but more about whether the test is checking on the "acceptance criteria" for a feature (i.e., *behaviour*), as visible to the user.

Feature tests should have a human-readable story that we can follow. We make it explicit using comments that accompany the test code. When creating a new FT, we can write the comments first, to capture the key points of the user story. Being human-readable, you could even share them with nonprogrammers, as a way of discussing the requirements and features of your app.

Test-driven development and Agile or Lean software development methodologies often go together, and one of the things we tend to talk about is the minimum viable app: what is the simplest thing we can build that is still useful? Let's start by building that, so that we can test the water as quickly as possible.

A minimum viable to-do list really only needs to let the user enter some to-do items, and remember them for their next visit.

Open up *functional_tests.py* and write a story a bit like this one:

functional_tests.py (ch02l001)

PYTHON

```python
from selenium import webdriver

browser = webdriver.Firefox()

# Edith has heard about a cool new online to-do app.
# She goes to check out its homepage
browser.get("http://localhost:8000")

# She notices the page title and header mention to-do lists
assert "To-Do" in browser.title

# She is invited to enter a to-do item straight away

# She types "Buy peacock feathers" into a text box
# (Edith's hobby is tying fly-fishing lures)

# When she hits enter, the page updates, and now the page lists
# "1: Buy peacock feathers" as an item in a to-do list

# There is still a text box inviting her to add another item.
# She enters "Use peacock feathers to make a fly" (Edith is very methodical)

# The page updates again, and now shows both items on her list

# Satisfied, she goes back to sleep

browser.quit()
```

We Have a Word for Comments…

When I first started at PythonAnywhere, I used to virtuously pepper my code with nice descriptive comments. My colleagues said to me: "Harry, we have a word for comments. We call them lies." I was shocked! I learned in school that comments are good practice?

They were exaggerating for effect. There is definitely a place for comments that add context and intention. But my colleagues were pointing out that comments aren't always as useful as you hope. For starters, it's pointless to write a comment that just repeats what you're doing with the code:

PYTHON

```python
# increment wibble by 1
wibble += 1
```

Not only is it pointless, but there’s a danger that you’ll forget to update the comments when you update the code, and they end up being misleading—lies! The ideal is to strive to make your code so readable, to use such good variable names and function names, and to structure it so well that you no longer need any comments to explain *what* the code is doing. Just a few here and there to explain *why*.

There are other places where comments are very useful. We’ll see that Django uses them a lot in the files it generates for us to use as a way of suggesting helpful bits of its API.

And, of course, we use comments to explain the user story in our functional tests—by forcing us to make a coherent story out of the test, it makes sure we’re always testing from the point of view of the user.

There is more fun to be had in this area, things like *Behaviour-Driven Development* (see Online Appendix: BDD (https://www.obeythetestinggoat.com/book/appendix_bdd.html)) and building domain-specific languages (DSLs) for testing, but they’re topics for other books.[2]

For more on comments, I recommend John Ousterhout’s *A Philosophy of Software Design*, which you can get a taste of by reading his lecture notes from the chapter on comments (https://oreil.ly/1cdgY).

You’ll notice that, apart from writing the test out as comments, I’ve updated the `assert` to look for "To-Do" instead of Django’s "Congratulations". That means we expect the test to fail now. Let’s try running it.

First, start up the server:

```
$ python manage.py runserver
```

And then, in another terminal, run the tests:

```
$ python functional_tests.py
Traceback (most recent call last):
  File "...goat-book/functional_tests.py", line 10, in <module>
    assert "To-Do" in browser.title
AssertionError
```

That’s what we call an *expected fail*, which is actually good news—not quite as good as a test that passes, but at least it’s failing for the right reason; we can have some confidence we’ve written the test correctly.

The Python Standard Library’s unittest Module

There are a couple of little annoyances we should probably deal with. Firstly, the message "AssertionError" isn’t very helpful—it would be nice if the test told us what it actually found as the browser title. Also, it’s left a Firefox window hanging around the desktop, so it would be nice if that got cleared up for us automatically.

One option would be to use the second parameter of the `assert` keyword, something like:

```python
assert "To-Do" in browser.title, f"Browser title was {browser.title}"
```

And we could also use `try/finally` to clean up the old Firefox window.

But these sorts of problems are quite common in testing, and there are some ready-made solutions for us in the standard library’s `unittest` module. Let’s use that! In *functional_tests.py*:

functional_tests.py (ch02l003)

PYTHON

```
import unittest
from selenium import webdriver


class NewVisitorTest(unittest.TestCase):  1
    def setUp(self):  3
        self.browser = webdriver.Firefox()  4

    def tearDown(self):  3
        self.browser.quit()

    def test_can_start_a_todo_list(self):  2
        # Edith has heard about a cool new online to-do app.
        # She goes to check out its homepage
        self.browser.get("http://localhost:8000")  4

        # She notices the page title and header mention to-do lists
        self.assertIn("To-Do", self.browser.title)  5

        # She is invited to enter a to-do item straight away
        self.fail("Finish the test!")  6

        [...]

        # Satisfied, she goes back to sleep


if __name__ == "__main__":  7
    unittest.main()  7
```

You'll probably notice a few things here:

1. Tests are organised into classes, which inherit from `unittest.TestCase`.

2. The main body of the test is in a method called `test_can_start_a_todo_list`. Any method whose name starts with `test_` is a test method, and will be run by the test runner. You can have more than one `test_` method per class. Nice descriptive names for our test methods are a good idea too.

3. `setUp` and `tearDown` are special methods that are run before and after each test. I'm using them to start and stop our browser. They're a bit like `try/finally`, in that `tearDown` will run even if there's an error during the test itself.[3] No more Firefox windows left lying around!

4 `browser`, which was previously a global variable, becomes `self.browser`, an attribute of the test class. This lets us pass it between `setUp`, `tearDown`, and the test method itself.

5 We use `self.assertIn` instead of just `assert` to make our test assertions. `unittest` provides lots of helper functions like this to make test assertions, like `assertEqual`, `assertTrue`, `assertFalse`, and so on. You can find more in the [`unittest` documentation](https://docs.python.org/3/library/unittest.html) (https://docs.python.org/3/library/unittest.html).

6 `self.fail` just fails no matter what, producing the error message given. I'm using it as a reminder to finish the test.

7 Finally, we have the `if __name__ == "__main__"` clause. (If you've not seen it before, that's how a Python script checks if it's been executed from the command line, rather than just imported by another script.) We call `unittest.main()`, which launches the `unittest` test runner, which will automatically find test classes and methods in the file and run them.

> If you've read the Django testing documentation, you might have seen something called `LiveServerTestCase`, and are wondering whether we should use it now. Full points to you for reading the friendly manual! `LiveServerTestCase` is a bit too complicated for now, but I promise I'll use it in a later chapter.

Let's try out our new and improved FT![4]

```
$ python functional_tests.py
F
======================================================================
FAIL: test_can_start_a_todo_list
(__main__.NewVisitorTest.test_can_start_a_todo_list)
 ---------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/functional_tests.py", line 18, in
test_can_start_a_todo_list
    self.assertIn("To-Do", self.browser.title)
    ~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 'To-Do' not found in 'The install worked successfully!
Congratulations!'

 ---------------------------------------------------------------------
Ran 1 test in 1.747s

FAILED (failures=1)
```

That's a bit nicer, isn't it? It tidied up our Firefox window, it gives us a nicely formatted report of how many tests were run and how many failed, and the `assertIn` has given us a helpful error message with useful debugging info. Bonzer!

> If you see some error messages saying `ResourceWarning` about "unclosed files", it's safe to ignore those. They seem to come and go, every few Selenium releases. They don't affect the important things to look for in our tracebacks and test results.

pytest Versus unittest

The Python world is increasingly turning from the standard-library provided `unittest` module towards a third-party tool called `pytest`. I'm a big fan too!

The Django project has a bunch of helpful tools designed to work with `unittest`. Although it is possible to get them to work with `pytest`, it felt like one thing too many to include in this book.

Read Brian Okken's Python Testing with pytest (https://pythontest.com/pytest-book) for an excellent, comprehensive guide to Pytest instead.

Commit

This is a good point to do a commit; it's a nicely self-contained change. We've expanded our functional test to include comments that describe the task we're setting ourselves, our minimum viable to-do list. We've also rewritten it to use the Python `unittest` module and its various testing helper functions.

Do a `git status`—that should assure you that the only file that has changed is *functional_tests.py*. Then do a `git diff -w`, which shows you the difference between the last commit and what's currently on disk, with the `-w` saying "ignore whitespace changes".

That should tell you that *functional_tests.py* has changed quite substantially:

```
$ git diff -w
diff --git a/functional_tests.py b/functional_tests.py
index d333591..b0f22dc 100644
--- a/functional_tests.py
+++ b/functional_tests.py
@@ -1,15 +1,24 @@
+import unittest
 from selenium import webdriver

-browser = webdriver.Firefox()

+class NewVisitorTest(unittest.TestCase):
+    def setUp(self):
+        self.browser = webdriver.Firefox()
+
+    def tearDown(self):
+        self.browser.quit()
+
+    def test_can_start_a_todo_list(self):
         # Edith has heard about a cool new online to-do app.
         # She goes to check out its homepage
-browser.get("http://localhost:8000")
+        self.browser.get("http://localhost:8000")

         # She notices the page title and header mention to-do lists
-assert "To-Do" in browser.title
+        self.assertIn("To-Do", self.browser.title)

         # She is invited to enter a to-do item straight away
+        self.fail("Finish the test!")

[...]
```

Now let's do a:

```
$ git commit -a
```

The `-a` means "automatically add any changes to tracked files" (i.e., any files that we've committed before). It won't add any brand new files (you have to explicitly `git add` them yourself), but often, as in this case, there aren't any new files, so it's a useful shortcut.

When the editor pops up, add a descriptive commit message, like "First FT specced out in comments, and now uses unittest".

Now that our FT uses a real test framework, and that we've got placeholder comments for what we want it to do, we're in an excellent position to start writing some real code for our lists app. Read on!

Useful TDD Concepts

User story

A description of how the application will work from the point of view of the user; used to structure a functional test

Expected failure

When a test fails in the way that we expected it to

1. If you want to read more about user stories, check out Gojko Adzic's *Fifty Quick Ideas to Improve Your User Stories* or Mike Cohn's *User Stories Applied: For Agile Software Development*.
2. Check out this video by the great Dave Farley if you want a taste: https://oreil.ly/bbawE.
3. The only exception is that if you have an exception inside `setUp`, then `tearDown` doesn't run.
4. Are you unable to move on because you're wondering what those *ch02l00x* things are, next to some of the code listings? They refer to specific commits (https://github.com/hjwp/book-example/commits/chapter_02_unittest) in the book's example repo. It's all to do with my book's own tests (https://github.com/hjwp/Book-TDD-Web-Dev-Python/tree/main/tests). You know, the tests for the tests in the book about testing. They have tests of their own, naturally.

Comments

25 Comments Kgotso Koete ▾

Join the discussion…

♡ 2 **Share** **Best** **Newest** **Oldest**

C

chris

6 years ago

Encountered an error attempting this with

geckodriver - 0.26.0
firefox - 72.0.1
python - 3.6.9
ubuntu - 18.04.3
django - 1.11.27

self.browser.quit() closed the firefox window, but then the test process would just hang -- I think a socket connection was hanging maybe?
Anyway, I spent about an hour trying to figure out what was going on and gave up and installed chromedriver. Example worked perfectly with chrome browser rather than FF.
I realize that's probably more of a geckodriver/FF issue than a problem with your book, just thought you might want to be made aware. Thanks for the book, so far its great!

1 0 Reply Share ›

J

John Qu

8 years ago

Hello, Harry. Found a typo error maybe.
In "note that they’re a bit like a `try/except`", do you mean `try/finally` as you mentioned above in "And we could also use a `try/finally` to clean up the old Firefox window."?
`tearDown()` is run anyway, this feather sound like what `finally` do, not `except` do.

0 0 Reply Share ›

hjwp Mod ➦ John Qu

7 years ago

quite right! will fix it, thanks John :)

0 0 Reply Share ›

This comment was deleted.

hjwp Mod ↪ Guest

8 years ago

you're on chapter 2 of a 26-chapter book. read on and you may find something more to your liking. but if you already have strong opinions about what the One True Way of tdd is, then this might not be the right book for you.

2 0 Reply Share ›

G

George Jetson ↪ Guest

7 years ago edited

Rafales If that was TDD, I'd find it easier to do. What everybody seems to say however is, TDD ALWAYS involves writing tests BEFORE code. To quote Uncle Bob : Red, Green, Re-factor. Writing small increments and then testing, is ordinary coding with good unit test coverage. To my understanding, Uncle Bob noticed, that people are lazy and frequently don't. However, based on chapters one and 2, I'm not seeing ever green tests. I assume that will be different, as I read on?

0 0 Reply Share ›

J

James Perry ↪ Guest

7 years ago edited

You are right. Why even bother reading this book. It is total garbage. The whole python and django web development community is wrong when they recommend this book. jk. Have some respect for the process guy!

0 0 Reply Share ›

hjwp Mod ↪ James Perry

7 years ago

lol. thanks for the support James. to be fair, people can have very different ways of doing TDD, and some really privilege the fast unit-test/code cycle that I describe later in the book (actually in the next chapter I think). But my initial intro using slow selenium tests can be quite offputting. i did think it was a bit unfair to throw the critique in so soon, but there you go. there's lots more discussion of the tradeoffs of different types of tests at the end of the book, in the last chapter in particular...

1 0 Reply Share ›

S

Sandy De La Rosa

8 years ago

I'm getting this error when I update the file "functional_test.py":
python functional_tests.py
File "functional_tests.py", line 22
[...rest of comments as before]
^
SyntaxError: invalid syntax

Why is this invalid syntax??

0 0 Reply Share ›

hjwp Mod ➦ Sandy De La Rosa

8 years ago

you're not meant to type that in literally - it's a placeholder to indicate that the rest of the file should contain the comments you already had before... I mention this in the intro chapter I think, but you'll see that placeholder [...] syntax in use a few times in the book, it's never meant to be typed in literally.

0 0 Reply Share ›

Jake Horvath

8 years ago

At the very end of the chapter when doing the git commit I kept getting an error message when i used:

git commit -a

It may be due to a git update or might just be something I did wrong, but I found the way to fix this was to do this:

git commit -a -m "First FT specced out in comments, and now uses unittest."

0 0 Reply Share ›

hjwp Mod ➦ Jake Horvath

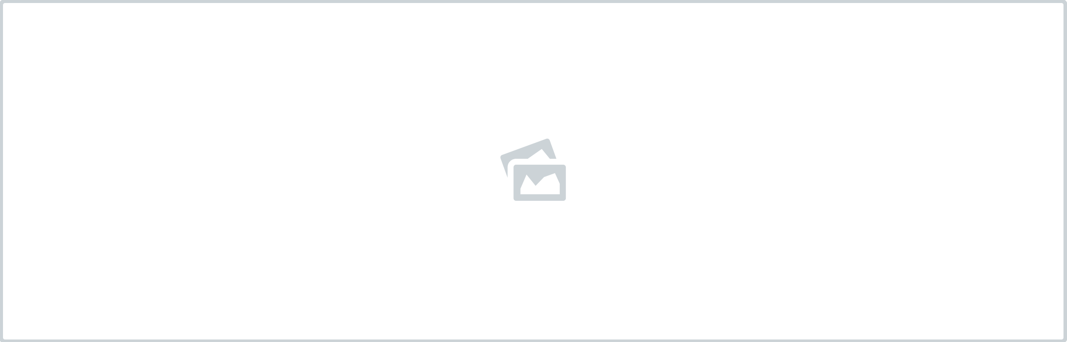
8 years ago

what error Jake? I wonder if it was to do with needing to configure your default editor for git. I mention that in the pre-reqs chapter...

0 0 Reply Share ›

Jake Horvath ➦ hjwp

8 years ago edited



I just changed a comment in the file to re-enact what happened. The picture is of "Git Bash". The command "git diff" isn't in the pic, but it works fine and shows the correct information. "E325 error" ,that says, "problem with the editor vi". It looks like it may have an issue with my "Vim" editor. I use "Atom.io" for this book, but sometimes use "Vim".

I probably just need to adjust some file in "Vim", but my initial comment was a shortcut to fix this error.

0 0 Reply Share ›

Jeremy Klein ➦ Jake Horvath —

8 years ago

It actually looks like it is looking for Vi instead of vim. If you would prefer to use atom as your default text editor you these instructions should help: https://help.github.com/art...

0 0 Reply Share ›

Zmie3 —

8 years ago

"This means that an FT can be a sort of specification for your application."

Shouldn't it be "a FT" not "an FT". Also if you run functional_test.py inside of superlists, than selenium will produce a geckodriver.log file which one should probably add to the .gitignore list. Not terribly important though.

0 0 Reply Share ›

hjwp Mod ➦ Zmie3 —

8 years ago

Hi there, the rules on an vs a before acronyms offer some room for using your judgement, but in general people tend to agree that you would write it the way you say it. https://english.stackexchan...

re: geckodriver.log, that should have been added to the .gitignore in the last chapter, you may have ended up with a slightly different folder structure to the one I'm assuming?

2 0 Reply Share ›

Zmie3 ➦ hjwp —

8 years ago

Oh, you're right. I started with a 1st edition version of the book and switched over to the newer one and totally missed that this was already fixed. My fault, sorry. Anyway really nice to read so far!

0 0 Reply Share ›

hjwp Mod ➦ Zmie3 —

8 years ago

thanks very much! please send more comments and suggestions!

1 0 Reply Share ›

VASUDHA BADDELI —

VASUDHA BADDELI

9 years ago

I follow the tutorial & i am getting this error:-
Traceback (most recent call last):
File "functional_test.py", line 4, in <module>
class NewVisitorTest(unittest.Testcase):
AttributeError: module 'unittest' has no attribute 'Testcase'

can some one help me out?

0 0 Reply Share ›

hjwp Mod ➦ VASUDHA BADDELI

9 years ago

check capitalization.

2 0 Reply Share ›

myFatherIsKing

9 years ago edited

I am following nicely so far. But I've run into problems in this chapter.
When I run my file, I get the following error in the test and tearDown methods

NewVisitorTest' object has no attribute 'browser'

I've tried deleting the "warnings='ignore'" line with no luck.
I don't know what could be the cause.

0 0 Reply Share ›

eznuh ➦ myFatherIsKing

8 years ago

I got the same error and noticed that I forgot to change "browser.get" to "self.browser.get". The error went away once I made that change.

0 0 Reply Share ›

myFatherIsKing ➦ myFatherIsKing

9 years ago

So I tinkered around. Luckily I had already figured out how to sync to GitHub.
Upon viewing the code on GitHub I noticed the indentation was all over the place, though it was very ok in my .py file.

So I completely re-wrote the script in a new file. No copy and paste used. Then the expected fail happened !!!

0 0 Reply Share ›

Aaron Nichols ➦ myFatherIsKing

9 years ago

at this point [illegible] attribute of the New Visitor Cl[illegible] to access it