Table of Contents

What Are We Doing with All These Tests? (And, Refactoring)

Now that we've seen the basics of TDD in action, it's time to pause and talk about why we're doing it.

I'm imagining several of you, dear readers, have been holding back some seething frustration—perhaps some of you have done a bit of unit testing before, and perhaps some of you are just in a hurry. You've been biting back questions like:

- Aren't all these tests a bit excessive?
- Surely some of them are redundant? There's duplication between the functional tests and the unit tests.
- Those unit tests seemed way too trivial—testing a one-line function that returns a constant! Isn't that just a waste of time? Shouldn't we save our tests for more complex things?

- What about all those tiny changes during the unit-test/code cycle? Couldn't we just skip to the end? I mean, `home_page = None` !? Really?
- You're not telling me you *actually* code like this in real life?

Ah, young grasshopper. I too was once full of questions like these. But only because they're perfectly good questions. In fact, I still ask myself questions like these—all the time. Does all this stuff really have value? Is this a bit of a cargo cult?

Programming Is Like Pulling a Bucket of Water Up from a Well

Ultimately, programming is hard. Often, we are smart, so we succeed. TDD is there to help us out when we're not so smart. Kent Beck (who basically invented TDD) uses the metaphor of lifting a bucket of water out of a well with a rope: when the well isn't too deep, and the bucket isn't very full, it's easy. And even lifting a full bucket is pretty easy at first. But after a while, you're going to get tired. TDD is like having a ratchet that lets you save your progress, so you can take a break, and make sure you never slip backwards.

That way, you don't have to be smart *all* the time (see Test ALL the things (adapted from Allie Brosh, Hyperbole and a Half)).

Figure 1. Test ALL the things (adapted from Allie Brosh, Hyperbole and a Half (https://oreil.ly/n_8R_))

OK, perhaps *in general*, you're prepared to concede that TDD is a good idea, but maybe you still think I'm overdoing it? Testing the tiniest thing, and taking ridiculously many small steps?

TDD is a *discipline*, and that means it's not something that comes naturally. Because many of the payoffs aren't immediate but only come in the longer term, you have to force yourself to do it in the moment. That's what the image of the Testing Goat is supposed to represent—you need to be a bit bloody-minded about it.

On the Merits of Trivial Tests for Trivial Functions

In the short term, it may feel a bit silly to write tests for simple functions and constants.

It's perfectly possible to imagine still doing "mostly" TDD, but following more relaxed rules where you don't unit test *absolutely* everything. But in this book my aim is to demonstrate full, rigorous TDD. Like a kata in a martial art, the idea is to learn the motions in a controlled context, when there is no adversity, so that the techniques are part of your muscle memory. It seems trivial now, because we've started with a very simple example. The problem comes when your application gets complex—that's when you really need your tests. And the danger is that complexity tends to sneak up on you, gradually. You may not notice it happening, but soon you're a boiled frog.

There are two other things to say in favour of tiny, simple tests for simple functions.

Firstly, if they're really trivial tests, then they won't take you that long to write. So stop moaning and just write them already.

Secondly, it's always good to have a placeholder. Having a test *there* for a simple function means it's that much less of a psychological barrier to overcome when the simple function gets a tiny bit more complex—perhaps it grows an `if`. Then a few weeks later, it grows a `for` loop. Before you know it, it's a recursive metaclass-based polymorphic tree parser factory. But because it's had tests from the very beginning, adding a new test each time has felt quite natural, and it's well tested. The alternative involves trying to decide when a function becomes "complicated enough", which is highly subjective. And worse, because there's no placeholder, it feels like that much more effort to start, so you're tempted each time to put it off…and pretty soon—frog soup!

Instead of trying to figure out some hand-wavy subjective rules for when you should write tests, and when you can get away with not bothering, I suggest following the discipline for now—and as with any discipline, you have to take the time to learn the rules before you can break them.

Now, let us return to our muttons.

Using Selenium to Test User Interactions

Where were we at the end of the last chapter? Let's rerun the test and find out:

```
$ 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 21, in
test_can_start_a_todo_list
    self.fail("Finish the test!")
    ~~~~~~~~~^^^^^^^^^^^^^^^^^^^^
AssertionError: Finish the test!

 ----------------------------------------------------------------------
Ran 1 test in 1.609s

FAILED (failures=1)
```

Did you try it, and get an error saying "Problem loading page" or "Unable to connect"? So did I. It's because we forgot to spin up the dev server first using `manage.py runserver`. Do that, and you'll get the failure message we're after.

> One of the great things about TDD is that you never have to worry about forgetting what to do next—just rerun your tests and they will tell you what you need to work on.

"Finish the test", it says, so let's do just that! Open up *functional_tests.py* and we'll extend our FT:

functional_tests.py (ch04l001)

PYTHON

```python
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
import time
import unittest


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
        self.browser.get("http://localhost:8000")

        # She notices the page title and header mention to-do lists
        self.assertIn("To-Do", self.browser.title)
        header_text = self.browser.find_element(By.TAG_NAME, "h1").text   1
        self.assertIn("To-Do", header_text)

        # She is invited to enter a to-do item straight away
        inputbox = self.browser.find_element(By.ID, "id_new_item")   1
        self.assertEqual(inputbox.get_attribute("placeholder"), "Enter a to-do
item")

        # She types "Buy peacock feathers" into a text box
        # (Edith's hobby is tying fly-fishing lures)
        inputbox.send_keys("Buy peacock feathers")   2

        # When she hits enter, the page updates, and now the page lists
        # "1: Buy peacock feathers" as an item in a to-do list table
        inputbox.send_keys(Keys.ENTER)   3
        time.sleep(1)   4

        table = self.browser.find_element(By.ID, "id_list_table")
        rows = table.find_elements(By.TAG_NAME, "tr")   1
        self.assertTrue(any(row.text == "1: Buy peacock feathers" for row in rows))

        # 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)
        self.fail("Finish the test!")
```

```
        # The page updates again, and now shows both items on her list
        [...]
```

1 We're using the two methods that Selenium provides to examine web pages: `find_element` and `find_elements` (notice the extra `s`, which means it will return several elements rather than just one). Each one is parameterised with a `By.SOMETHING`, which lets us search using different HTML properties and attributes.

2 We also use `send_keys`, which is Selenium's way of typing into input elements.

3 The `Keys` class (don't forget to import it) lets us send special keys like Enter.[1]

4 When we hit Enter, the page will refresh. The `time.sleep` is there to make sure the browser has finished loading before we make any assertions about the new page. This is called an "explicit wait" (a very simple one; we'll improve it in [chapter_06_explicit_waits_1]).

Watch out for the difference between the Selenium `find_element()` and `find_elements()` functions. One returns an element and raises an exception if it can't find it, whereas the other returns a list, which may be empty.

Also, just look at that `any()` function. It's a little-known Python built-in. I don't even need to explain it, do I? Python is such a joy.[2]

If you're one of my readers who doesn't know Python, what's happening *inside* the `any()` may need some explaining. The basic syntax is that of a *list comprehension*, and if you haven't learned about them, you should do so immediately! Trey Hunner's explanation is excellent (https://oreil.ly/6bX0h). In point of fact, because we're omitting the square brackets, we're actually using a *generator expression* rather than a list comprehension. It's probably less important to understand the difference between those two, but if you're curious, Guido van Rossum, the inventor of Python, has written a blog post explaining the difference (https://oreil.ly/Om6vK).

Let's see how it gets on:

```
$ python functional_tests.py
[...]
  File "...goat-book/functional_tests.py", line 22, in
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: h1; For documentation on this error, please visit: [...]
```

Decoding that, the test is saying it can't find an `<h1>` element on the page. Let's see what we can do to add that to the HTML of our home page.

Big changes to a functional test are usually a good thing to commit on their own. I failed to do so when I was first working out the code for this chapter, and I regretted it later when I changed my mind and had the change mixed up with a bunch of others. The more atomic your commits, the better:

```
$ git diff  # should show changes to functional_tests.py
$ git commit -am "Functional test now checks we can input a to-do item"
```

The "Don't Test Constants" Rule, and Templates to the Rescue

Let's take a look at our unit tests, *lists/tests.py*. Currently we're looking for specific HTML strings, but that's not a particularly efficient way of testing HTML. In general, one of the rules of unit testing is "don't test constants", and testing HTML as text is a lot like testing a constant.

In other words, if you have some code that says:

PYTHON

```
wibble = 3
```

There's not much point in a test that says:

PYTHON

```
from myprogram import wibble
assert wibble == 3
```

Unit tests are really about testing logic, flow control, and configuration. Making assertions about exactly what sequence of characters we have in our HTML strings isn't doing that.

It's not *quite* that simple, because HTML is code after all, and we do want something to check that we've written code that works—but that's our FT's job, not the unit test's.

So maybe "don't test constants" isn't the online guideline at play here, but in any case, mangling raw strings in Python really isn't a great way of dealing with HTML. There's a much better solution, which is to use templates. Quite apart from anything else, if we can keep HTML to one side in a file whose name ends in *.html*, we'll get better syntax highlighting!

There are lots of Python templating frameworks out there, and Django has its own which works very well. Let's use that.

Refactoring to Use a Template

What we want to do now is make our view function return exactly the same HTML, but just using a different process. That's a refactor—when we try to improve the code *without changing its functionality*.

That last bit is really important. If you try to add new functionality at the same time as refactoring, you're much more likely to run into trouble. Refactoring is actually a whole discipline in itself, and it even has a reference book: Martin Fowler's *Refactoring* (http://refactoring.com).

The first rule is that you can't refactor without tests. Thankfully, we're doing TDD, so we're way ahead of the game. Let's check that our tests pass; they will be what makes sure that our refactoring is behaviour-preserving:

```
$ python manage.py test
[...]
OK
```

Great! We'll start by taking our HTML string and putting it into its own file. Create a directory called *lists/templates* to keep templates in, and then open a file at *lists/templates/home.html*, to which we'll transfer our HTML:[3]

lists/templates/home.html (ch04l002)

```
<html>
  <title>To-Do lists</title>
</html>
```

HTML

Mmm, syntax-highlighted…much nicer! Now to change our view function:

lists/views.py (ch04l003)

```python
from django.shortcuts import render


def home_page(request):
    return render(request, "home.html")
```

Instead of building our own `HttpResponse`, we now use the Django `render()` function. It takes the request as its first parameter (for reasons we'll go into later) and the name of the template to render. Django will automatically search folders called *templates* inside any of your apps' directories. Then it builds an `HttpResponse` for you, based on the content of the template.

> Templates are a very powerful feature of Django's, and their main strength consists of substituting Python variables into HTML text. We're not using this feature yet, but we will in future chapters. That's why we use `render()` rather than, say, manually reading the file from disk with the built-in `open()`.

Let's see if it works:

```
$ python manage.py test
[...]
======================================================================
ERROR: test_home_page_returns_correct_html
(lists.tests.HomePageTest.test_home_page_returns_correct_html)   2
----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/lists/tests.py", line 7, in test_home_page_returns_correct_html
    response = self.client.get("/")   3
               ^^^^^^^^^^^^^^^^^^^^
[...]
  File "...goat-book/lists/views.py", line 4, in home_page
    return render(request, "home.html")   4
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File ".../django/shortcuts.py", line 24, in render
    content = loader.render_to_string(template_name, context, request, using=using)
              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File ".../django/template/loader.py", line 61, in render_to_string
    template = get_template(template_name, using=using)
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File ".../django/template/loader.py", line 19, in get_template
    raise TemplateDoesNotExist(template_name, chain=chain)
django.template.exceptions.TemplateDoesNotExist: home.html   1

----------------------------------------------------------------------
Ran 1 test in 0.074s
```

Another chance to analyse a traceback:

1. We start with the error: it can't find the template.
2. Then we double-check what test is failing: sure enough, it's our test of the view HTML.
3. Then we find the line in our tests that caused the failure: it's when we request the root URL ("/").
4. Finally, we look for the part of our own application code that caused the failure: it's when we try to call `render`.

So why can't Django find the template? It's right where it's supposed to be, in the *lists/templates* folder.

The thing is that we haven't yet *officially* registered our lists app with Django. Unfortunately, just running the `startapp` command and having what is obviously an app in your project folder isn't quite enough. You have to tell Django that you *really* mean it, and add it to *settings.py* as well

—belt and braces. Open it up and look for a variable called `INSTALLED_APPS`, to which we'll add `lists`:

superlists/settings.py (ch04l004)

```
# Application definition

INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "lists",
]
```

PYTHON

You can see there's lots of apps already in there by default. We just need to add ours to the bottom of the list. Don't forget the trailing comma—it may not be required, but one day you'll be really annoyed when you forget it and Python concatenates two strings on different lines…

Now we can try running the tests again:

```
$ python manage.py test
[...]
OK
```

And we can double-check with the FTs:

```
$ python functional_tests.py
[...]
  File "...goat-book/functional_tests.py", line 22, in
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: h1; For documentation on this error, please visit: [...]
```

Good, they still get to the same place they did before. Our refactor of the code is now complete, and the tests mean we're happy that behaviour is preserved. Now we can change the tests so that they're no longer testing constants; instead, they should just check that we're rendering the right template.

Revisiting Our Unit Tests

Our unit tests are currently essentially checking HTML by hand—certainly that's very close to "testing constants".

lists/tests.py

```
def test_home_page_returns_correct_html(self):
    response = self.client.get("/")
    self.assertContains(response, "<title>To-Do lists</title>")   1
    self.assertContains(response, "<html>")
    self.assertContains(response, "</html>")
```

PYTHON

We don't want to be duplicating the full content of our HTML template in our tests, or even last sections of it. What could we do instead?

Rather than testing the full template, we could just check that we're using the *right* template. The Django test client has a method, `assertTemplateUsed` , which will let us do just that.

lists/tests.py (ch04l005)

```
def test_home_page_returns_correct_html(self):
    response = self.client.get("/")
    self.assertContains(response, "<title>To-Do lists</title>")   1
    self.assertContains(response, "<html>")
    self.assertContains(response, "</html>")
    self.assertTemplateUsed(response, "home.html")   2
```

PYTHON

1 We'll leave the old tests there for now, just to make sure everything is working the way we think it is.

2 `.assertTemplateUsed` lets us check what template was used to render a response. (NB: It will only work for responses that were retrieved by the test client.)

And that test will still pass:

```
Ran 1 tests in 0.016s

OK
```

Just because I'm always suspicious of a test I haven't seen fail, let's deliberately break it:

lists/tests.py (ch04l006)

```
    self.assertTemplateUsed(response, "wrong.html")
```
PYTHON

That way, we'll also learn what its error messages look like:

```
AssertionError: False is not true : Template 'wrong.html' was not a template
used to render the response. Actual template(s) used: home.html
```

That's very helpful!

Let's change the assert back to the right thing.

lists/tests.py (ch04l007)

```
from django.test import TestCase


class HomePageTest(TestCase):
    def test_uses_home_template(self):
        response = self.client.get("/")
        self.assertTemplateUsed(response, "home.html")
```
PYTHON

Now, instead of testing constants we're testing at a higher level of abstraction. Great!

Test Behaviour, Not Implementation

As so often in the world of programming though, things are not black and white.

Yes, on the plus side, our tests no longer care about the specific content of our HTML so they are no longer brittle with respect to minor changes of the copy in our template.

But on the other hand, they depend on some Django implementation details, so they *are* brittle with respect to changing the template rendering library, or even just renaming templates.

In a way, testing for the template name (and implicitly, even checking that we used a template at all) is a lot like testing implementation. So what is the *behaviour* that we want?

Yes, in a sense, the "behaviour" we want from the view is "render the template". But from the point of view of the user, it's "show me the home page".

We're also vulnerable to accidentally breaking the template. Let's try it now, by just deleting all the contents of the template file:

```
$ mv lists/templates/home.html lists/templates/home.html.bak
$ touch lists/templates/home.html
$ python manage.py test
[...]
OK
```

Yes, our FTs will pick up on this, so ultimately we're OK:

```
$ python functional_tests.py
[...]
    self.assertIn("To-Do", self.browser.title)
    ~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 'To-Do' not found in ''
```

But it would be nice to have our unit tests pick up on this too:

```
$ mv lists/templates/home.html.bak lists/templates/home.html
```

Deciding exactly what to test with FTs and what to test with unit tests is a fine line, and the objective is not to double-test everything. But in general, the more we can test with unit tests the better. They run faster, and they give more specific feedback.

So, let's bring back a minimal "smoke test"[4] to check that what we're rendering is actually the home page:

lists/tests.py (ch04l008)

```
PYTHON
class HomePageTest(TestCase):
    def test_uses_home_template(self):
        response = self.client.get("/")
        self.assertTemplateUsed(response, "home.html")   1

    def test_renders_homepage_content(self):
        response = self.client.get("/")
        self.assertContains(response, "To-Do")   2
```

1 We'll keep this first test, which asserts on whether we're rendering the right "constant".

2 And this gives us a minimal smoke test that we have got the right content in the template.

As our home page template gains more functionality over the next couple of chapters, we'll come back to talking about what to test here in the unit tests and what to leave to the FTs.

Unit tests give you faster and more specific feedback than FTs. Bear this in mind when deciding what to test where.

We'll visit the trade-offs between different types of tests at several points in the book, and particularly in [chapter_27_hot_lava].

On Refactoring

That was an absolutely trivial example of refactoring. But, as Kent Beck puts it in *Test-Driven Development: By Example*, "Am I recommending that you actually work this way? No. I'm recommending that you be *able* to work this way".

In fact, as I was writing this my first instinct was to dive in and change the test first—make it use the `assertTemplateUsed()` function straight away; against the expected render; and then go ahead and make the code change. But notice how that actually would have left space for me to break things: I could have defined the template as containing *any* arbitrary string, instead of the string with the right `<html>` and `<title>` tags.

When refactoring, work on either the code or the tests, but not both at once.

There's always a tendency to skip ahead a couple of steps, to make a few tweaks to the behaviour while you're refactoring. But pretty soon you've got changes to half a dozen different files, you've totally lost track of where you are, and nothing works anymore. If you don't want to end up like Refactoring Cat (https://oreil.ly/F_Hqf) (Refactoring Cat—be sure to look up the full animated GIF (source: 4GIFs.com)), stick to small steps; keep refactoring and functionality changes entirely separate.

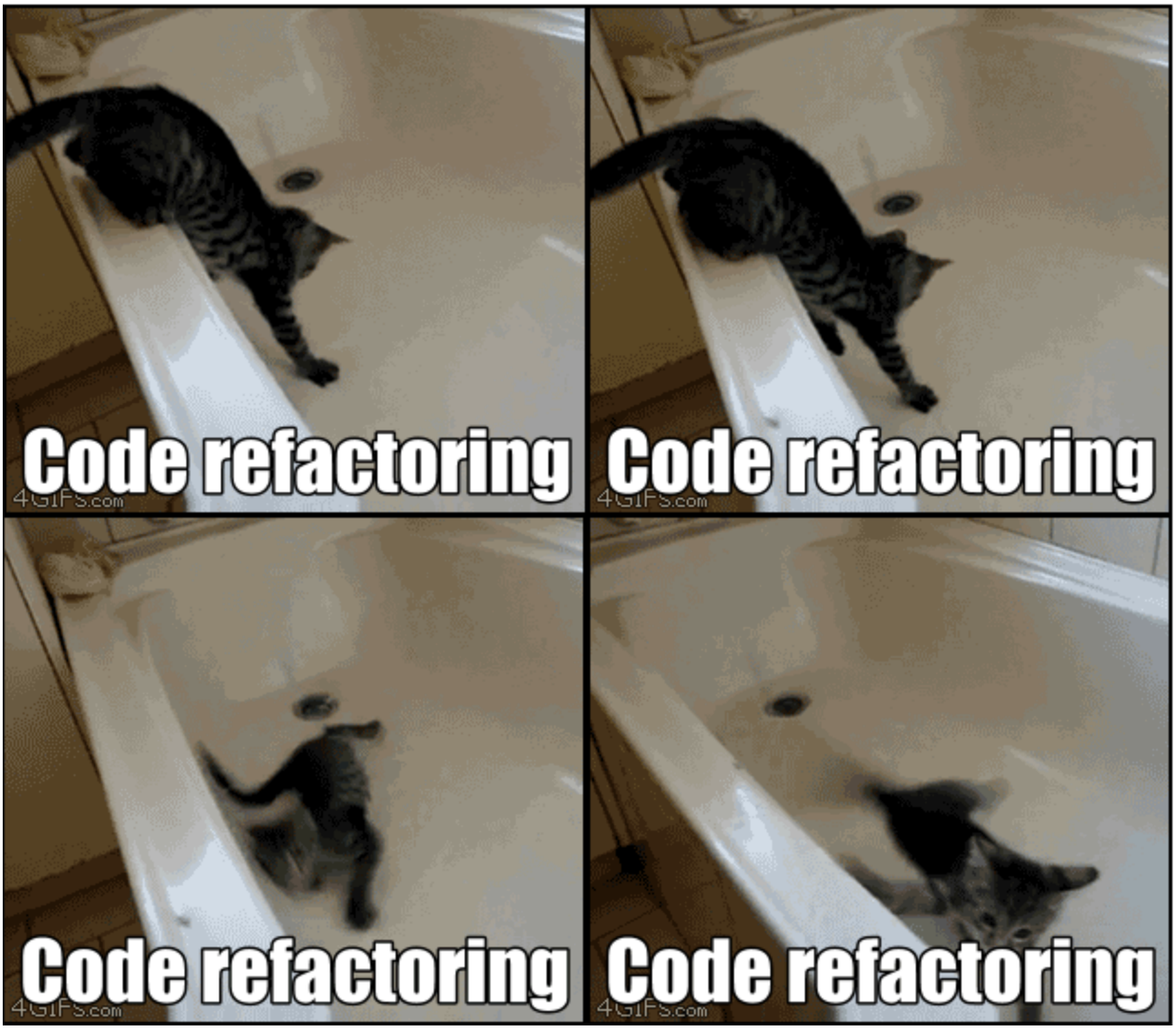

Figure 2. Refactoring Cat—be sure to look up the full animated GIF (source: 4GIFs.com)

We'll come across Refactoring Cat again during this book, as an example of what happens when we get carried away and change too many things at once. Think of it as the little cartoon demon counterpart to the Testing Goat, popping up over your other shoulder and giving you bad advice.

It's a good idea to do a commit after any refactoring:

```
$ git status # see tests.py, views.py, settings.py, + new templates folder
$ git add .  # will also add the untracked templates folder
$ git diff --staged # review the changes we're about to commit
$ git commit -m "Refactor homepage view to use a template"
```

A Little More of Our Front Page

In the meantime, our FT is still failing. Let's now make an actual code change to get it passing. Because our HTML is now in a template, we can feel free to make changes to it, without needing to write any extra unit tests.

> This is another distinction between FTs and unit tests; because the FTs use a real web browser, we use them as the primary tool for testing our UI, and the HTML that implements it.

So, we wanted an `<h1>`:

lists/templates/home.html (ch04l009)

```
HTML
<html>
  <head>
    <title>To-Do lists</title>
  </head>
  <body>
    <h1>Your To-Do list</h1>
  </body>
</html>
```

Let's see if our FT likes it a little better:

```
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_new_item"]; For documentation on this error, [...]
```

OK, let's add an input with that ID:

lists/templates/home.html (ch04l010)

```
    [...]
    <body>
      <h1>Your To-Do list</h1>
      <input id="id_new_item" />
    </body>
  </html>
```

HTML

And now what does the FT say?

```
AssertionError: '' != 'Enter a to-do item'
```

We add our placeholder text...

lists/templates/home.html (ch04l011)

```
      <input id="id_new_item" placeholder="Enter a to-do item" />
```

HTML

Which gives:

```
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_list_table"]; [...]
```

So we can go ahead and put the table onto the page. At this stage it'll just be empty:

lists/templates/home.html (ch04l012)

```
      <input id="id_new_item" placeholder="Enter a to-do item" />
      <table id="id_list_table">
      </table>
    </body>
```

HTML

What does the FT think?

```
[...]
  File "...goat-book/functional_tests.py", line 40, in
test_can_start_a_todo_list
    self.assertTrue(any(row.text == "1: Buy peacock feathers" for row in rows))
    ~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: False is not true
```

Slightly cryptic! We can use the line number to track it down, and it turns out it's that `any()` function I was so smug about earlier—or, more precisely, the `assertTrue`, which doesn't have a very explicit failure message. We can pass a custom error message as an argument to most `assertX` methods in `unittest`:

functional_tests.py (ch04l013)

```
        self.assertTrue(
            any(row.text == "1: Buy peacock feathers" for row in rows),
            "New to-do item did not appear in table",
        )
```

PYTHON

If you run the FT again, you should see our helpful message:

```
AssertionError: False is not true : New to-do item did not appear in table
```

But now, to get this to pass, we will need to actually process the user's form submission. And that's a topic for the next chapter.

For now let's do a commit:

```
$ git diff
$ git commit -am "Front page HTML now generated from a template"
```

Thanks to a bit of refactoring, we've got our view set up to render a template, we've stopped testing constants, and we're now well placed to start processing user input.

Recap: The TDD Process

We've now seen all the main aspects of the TDD process, in practice:

- Functional tests
- Unit tests
- The unit-test/code cycle
- Refactoring

It's time for a little recap, and perhaps even some flowcharts. (Forgive me, my years misspent as a management consultant have ruined me. On the plus side, said flowcharts will feature recursion!)

What does the overall TDD process look like?

- We write a test.
- We run the test and see it fail.
- We write some minimal code to get it a little further.
- We rerun the test and repeat until it passes (the unit-test/code cycle)
- Then, we look for opportunities to refactor our code, using our tests to make sure we don't break anything.
- Then, we look for opportunities to refactor our tests too, while attempting to stick to rules like "test behaviour, not implementation" and "don't test constants".
- And start again from the top!

See TDD process as a flowchart, including the unit-test/code cycle.

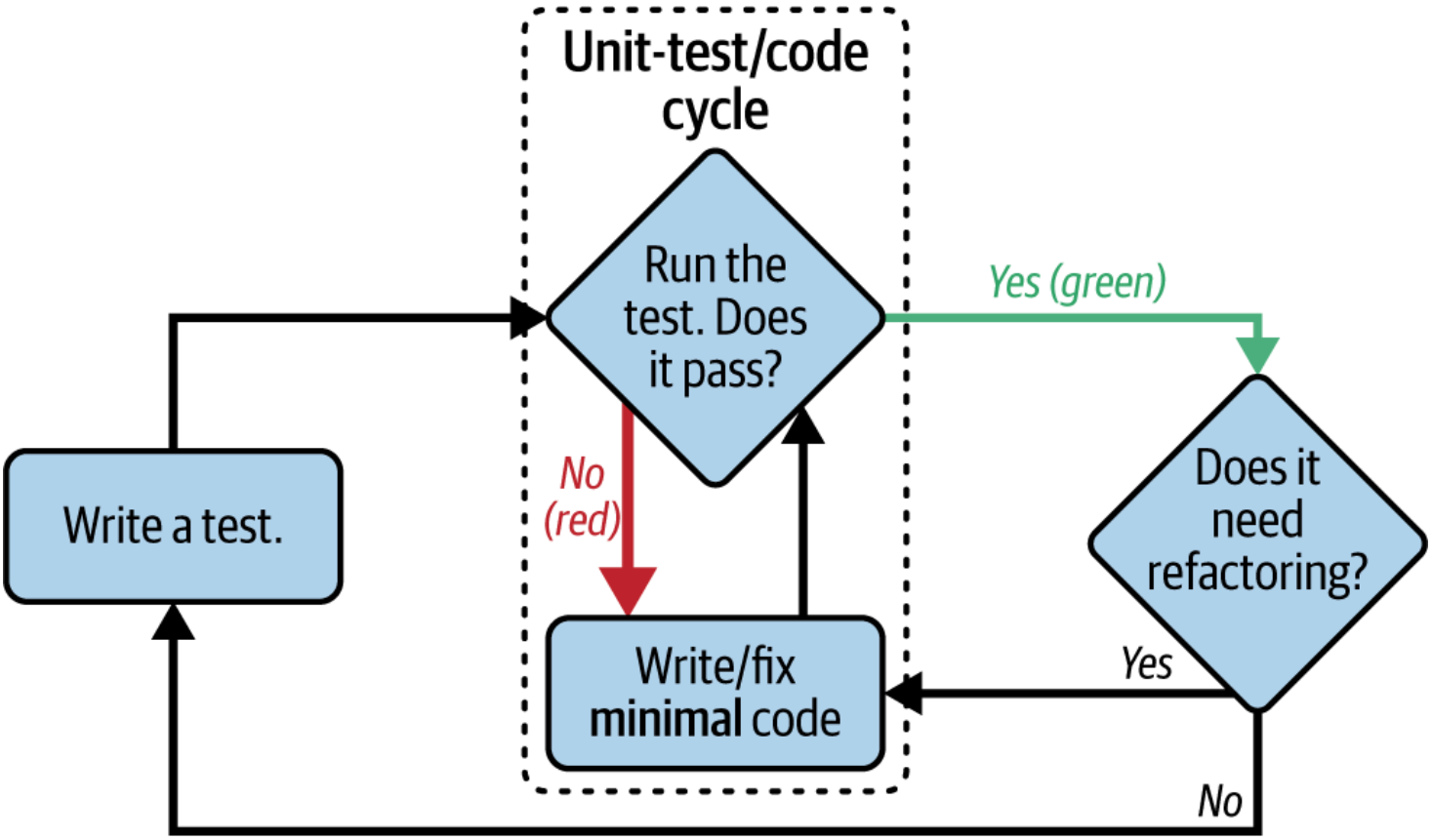

Figure 3. TDD process as a flowchart, including the unit-test/code cycle

It's very common to talk about this process using the three words: *red/green/refactor*. See Red/green/refactor.

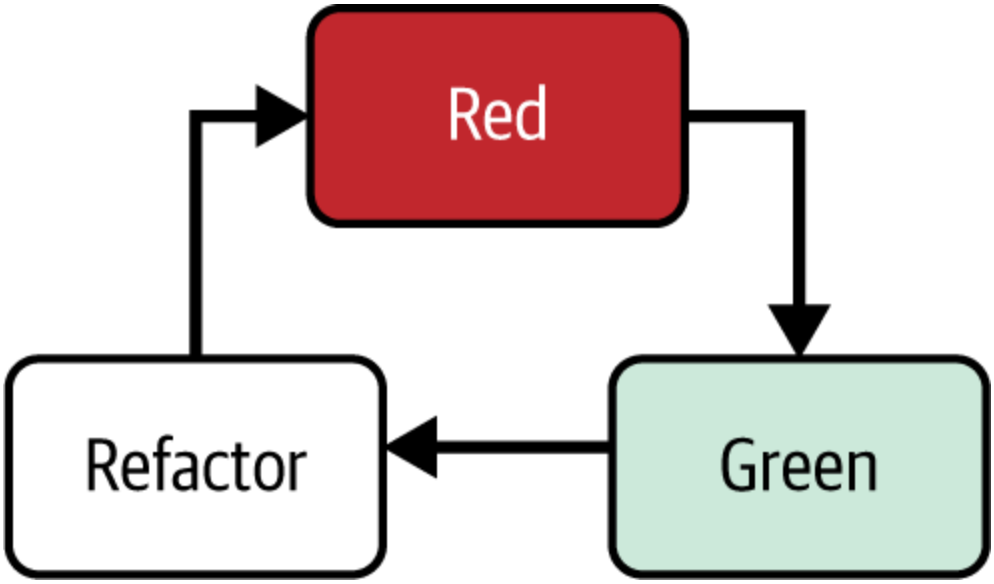

Figure 4. Red/green/refactor

- We write a test, and see it fail ("red").
- We cycle between code and tests until the test passes ("green").

- Then, we look for opportunities to refactor.
- Repeat as required!

Double-loop TDD

But how does this apply when we have functional tests *and* unit tests? Well, you can think of the FT as driving a higher-level version of the same cycle, with an inner red/green/refactor loop being required to get an FT from red to green; see Double-loop TDD: Inner and outer loops.

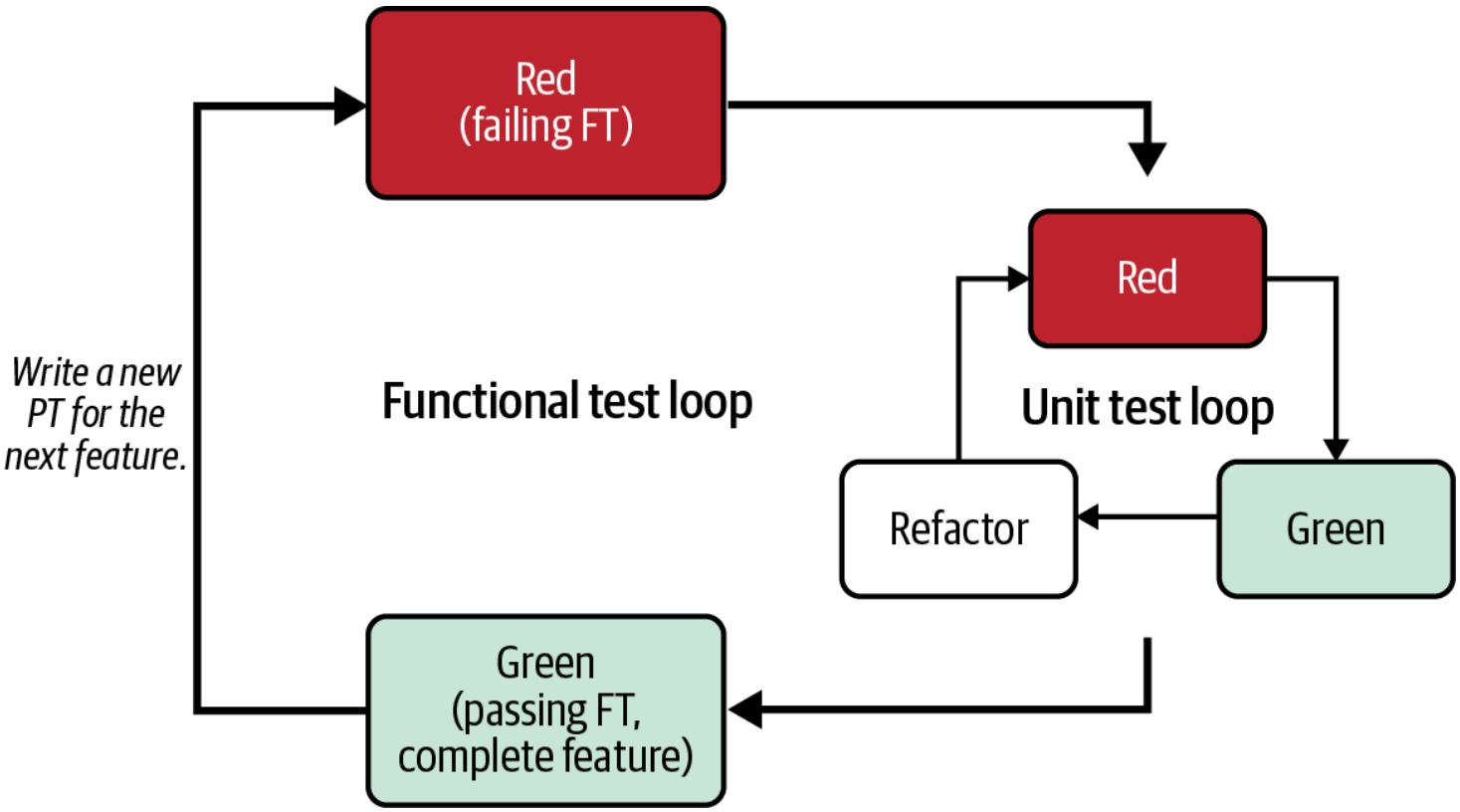

Figure 5. Double-loop TDD: Inner and outer loops

When a new feature or business requirement comes along, we write a new (failing) FT to capture a high-level view of the requirement. It may not cover every last edge case, but it should be enough to reassure ourselves that things are working.

To get that FT to green, we then enter into the lower-level unit test cycle, where we put together all the moving parts required, and add tests for all the edge cases. Any time we get to green and refactored at the unit test level, we can pop back up to the FT level to guide us towards the next thing we need to work on. Once both levels are green, we can do any extra refactoring or work on edge cases.

We'll explore all of the different parts of this workflow in more detail over the coming chapters.

How to "Check" Your Code, or Skip Ahead (If You Must)

All of the code examples I've used in the book are available in my repo on GitHub (https://github.com/hjwp/book-example/). So, if you ever want to compare your code against mine, you can take a look at it there.

Each chapter has its own branch, which is named after its short name. The one for this chapter (https://github.com/hjwp/book-example/tree/chapter_04_philosophy_and_refactoring) is a snapshot of the code as it should be at the *end* of the chapter.

You can find a full list of them in [appendix_github_links], as well as instructions on how to download them or use Git to compare your code to mine.

Obviously I can't possibly condone it, but you can also use my repo to "skip ahead" and check out the code to let you work on a later chapter without having worked through all the earlier chapters yourself. You're only cheating yourself you know!

---

1. You could also just use the string `"\n"`, but `Keys` also lets you send special keys like Ctrl, so I thought I'd show it.
2. Python *is* most definitely a joy, but if you think I'm being a bit smug here, I don't blame you! Actually, I wish I'd picked up on this feeling of self-satisfaction and seen it as a warning sign that I was being a little *too* clever. In the next chapter, you'll see I get my comeuppance.
3. Some people like to use another subfolder named after the app (i.e., *lists/templates/lists*) and then refer to the template as *lists/home.html*. This is called "template namespacing". I figured it was overcomplicated for this small project, but it may be worth it on larger projects. There's more in the Django tutorial (https://docs.djangoproject.com/en/5.2/intro/tutorial03/#write-views-that-actually-do-something).
4. A smoke test is a minimal test that can quickly tell you if something is wrong, without exhaustively testing every aspect that you might care about. Wikipedia has some fun speculation on the etymology (https://oreil.ly/1_isl).

Comments

ALSO ON **OBEY THE TESTING GOAT!**

Making our App Production-Ready

2 years ago • 8 comments

Static files aren't the same kind of things as the dynamic content that comes from …

Getting to a Production-Ready Deployment

9 years ago • 99 comments

Our deployment is working fine but it's not production-ready. Let's try to get it …

Finishing "My Lists": Outside-In TDD

2 years ago • 3 comments

The @property Decorator in Python This is a powerful feature of the language, …

Getting A For Deplo

a year ago • 1

With a PaaS your own se renting a "se

Emoji?

7 Responses

 Upvote  Funny  Love  Surprised  Angry  Sad

5 Comments Kgotso Koete ▾

Join the discussion…

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

D **Deepstop**

2 years ago edited

I have a hard time not duplicating functional tests in the unit tests in double-loop tdd. Say the functional test has check the expected result in a table row, so it has to navigate down the table hierarchy - table tag, row tag, td tag. The table is not started yet, so create a unit test that looks for the table (i.e., with BeautifulSoup). That fails too, so create <table></table> in the page. Now the unit test passes and the functional test goes a little further. Repeat. The user story gets lost in the navigation.

It would seem better to me if there were some common way to navigate in both cases, so that the functional test could concentrate on functionality, and the unit tests could handle the "get this bit from that element" stuff for not only testing, but also on behalf of the functional test.

Otherwise I think I'd let the functional tests worry about elements having all the required bits (beyond the html tags and correct title) and leave the unit testing to business logic.

2 0 Reply Share ›

H **Heatherirene** ↗ Deepstop

8 months ago

Hello Deepstop,

This bit, "I have a hard time not duplicating functional tests in the unit tests in double-loop tdd," I think is one of the banes every tester faces. I've read from some testing experts it's better to repeat your code for clarity in tests. Sometimes I agree with this sentiment, and sometimes I don't. Sometimes repetition is unavoidable in testing.

In regards to this comment, "It would seem better to me if there were some common way to navigate in both cases." There is! Put ids in your html (once you have html) -- even down to the table cells, if possible. Put ids on everything.

This author has neglected to mention this method, and is using vague finds by tag and text. Such finds make for brittle tests. Tags and text can change, sometimes frequently. Now, your find by tag or text test is broken. Find by id is much more reliable, faster, and precise (no traversing the table). This author should not only be using it for almost everything in the html, but also should be espousing using ids in all of your html always.

Happy testing!

Heather

1 0 Reply Share ›

H

Heatherirene

8 months ago

Please, hire an editor ASAP. Your writing is, to write the least, not good -- both in grammar/punctuation and in tone. Not everything should be given emphasis [with exclamation points]! The texts are riddled with run-on sentences. Also, you're long-winded on most of your anecdotes and analogies, and then, overly vague on important technicalities. Please, edit for clarity and detail.

For instance, you write to be sure to import Keys in this chapter, but then neglect to write which Keys, from where? The sentence is even footnoted with a useless footnote. Footnotes should be reserved for important information, like which Keys to import, from where, not useless factoids anyone can look up on the internet, StackOverflow, and now AI. I had to look up with AI which Keys you're writing about here because there are too many from which to choose to write simply "import Keys".

Also, did you write this book using AI, like Copilot? Did the AI actually write this book? (It would explain much of the poor writing, like too much being emphasized.) I started following along with the functional test's expansion in VS Code with Copilot on, and Copilot wrote the exact same test, including the same comments, minus the peacock feathers for the one exception. Copilot wrote: "# User types 'Buy milk' into the input box." Copilot even has the same vague variable names, e.g.: "inputbox". (Whereas, best test practice usually is to name variables more precisely for what they are, e.g.: textbox, radio_input/radio_btn, select/dropdown, etc. These examples are all inputs, but they're not the same kind of input.)

0 0 Reply Share ›

D

disqus_Q7gnQr5QwP ➚ Heatherirene

4 months ago