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

<http://www.obeythetestinggoat.com/book/chapter_philosophy_and_refactoring.html>

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.
* I mean, what are you doing importing django.urls.resolve in your unit tests? Isn’t that testing Django—​that is, testing third-party code? I thought that was a no-no?
* Those unit tests seemed way too trivial—​testing one line of declaration, and 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? Surely we could have just skipped 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, take a break, and make sure you never slip backwards. That way you don’t have to be smart *all* the time.



Figure 1. Test ALL the things (original illustration source: [Allie Brosh, Hyperbole and a Half](http://bit.ly/1iXxdYp))

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 illustrate—​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 techiques 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 quite 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 them. 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, but worse, because there’s no placeholder, it seems like that much more effort, and you’re tempted each time to put it off a little longer, 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—​as with any discipline, you have to take the time to learn the rules before you can break them.

Now, back to our onions.

**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\_list\_and\_retrieve\_it\_later (\_\_main\_\_.NewVisitorTest)

---------------------------------------------------------------------

Traceback (most recent call last):

File "functional\_tests.py", line 19, in

test\_can\_start\_a\_list\_and\_retrieve\_it\_later

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

from selenium import webdriver

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\_list\_and\_retrieve\_it\_later(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

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')

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')

# 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)

time.sleep(1)

table = self.browser.find\_element\_by\_id('id\_list\_table')

rows = table.find\_elements\_by\_tag\_name('tr')

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

[...]

|  |  |
| --- | --- |
|  | We’re using several of the methods that Selenium provides to examine web pages: find\_element\_by\_tag\_name, find\_element\_by\_id, and find⁠\_ele⁠ment⁠**s**⁠\_by⁠\_​tag\_name (notice the extra s, which means it will return several elements rather than just one). |
|  | We also use send\_keys, which is Selenium’s way of typing into input elements. |
|  | The Keys class (don’t forget to import it) lets us send special keys like Enter.[[1](http://www.obeythetestinggoat.com/book/chapter_philosophy_and_refactoring.html#_footnote_1)] |
|  | 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\_explicit\_waits\_1]](http://www.obeythetestinggoat.com/book/chapter_explicit_waits_1.html)). |
|  | 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.

Although, if you’re one of my readers who doesn’t know Python, what’s happening inside the any is a *generator expression*, which is like a *list comprehension* but awesomer. You need to read up on this. If you Google it, you’ll find [Guido himself explaining it nicely](http://bit.ly/1iXxD18). Come back and tell me that’s not pure joy!

Let’s see how it gets on:

$ **python functional\_tests.py**

[...]

selenium.common.exceptions.NoSuchElementException: Message: Unable to locate

element: h1

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 in my first draft, 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:

wibble = 3

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

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.

What’s more, 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:[[2](http://www.obeythetestinggoat.com/book/chapter_philosophy_and_refactoring.html#_footnote_2)]

lists/templates/home.html

<html>

<title>To-Do lists</title>

</html>

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

lists/views.py

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 and (later) render\_to\_ string 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)

---------------------------------------------------------------------

Traceback (most recent call last):

File "...python-tdd-book/lists/tests.py", line 17, in

test\_home\_page\_returns\_correct\_html

response = home\_page(request)

File "...python-tdd-book/lists/views.py", line 5, in home\_page

return render(request, 'home.html')

File "/usr/local/lib/python3.6/dist-packages/django/shortcuts.py", line 48,

in render

return HttpResponse(loader.render\_to\_string(\*args, \*\*kwargs),

File "/usr/local/lib/python3.6/dist-packages/django/template/loader.py", line

170, in render\_to\_string

t = get\_template(template\_name, dirs)

File "/usr/local/lib/python3.6/dist-packages/django/template/loader.py", line

144, in get\_template

template, origin = find\_template(template\_name, dirs)

File "/usr/local/lib/python3.6/dist-packages/django/template/loader.py", line

136, in find\_template

raise TemplateDoesNotExist(name)

django.template.base.TemplateDoesNotExist: home.html

---------------------------------------------------------------------

Ran 2 tests in 0.004s

Another chance to analyse a traceback:

|  |  |
| --- | --- |
|  | We start with the error: it can’t find the template. |
|  | Then we double-check what test is failing: sure enough, it’s our test of the view HTML. |
|  | Then we find the line in our tests that caused the failure: it’s when we call the home\_page function. |
|  | 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

# Application definition

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'lists',

]

You can see there’s lots of apps already in there by default. We just need to add ours, lists, 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**

[...]

self.assertTrue(html.endswith('</html>'))

AssertionError: False is not true

Darn, not quite.

|  |  |
| --- | --- |
|  | Depending on whether your text editor insists on adding newlines to the end of files, you may not even see this error. If so, you can safely ignore the next bit, and skip straight to where you can see the listing says OK. |

But it did get further! It seems it managed to find our template, but the last of the three assertions is failing. Apparently there’s something wrong at the end of the output. I had to do a little print(repr(html)) to debug this, but it turns out that the switch to templates has introduced an additional newline (\n) at the end. We can get them to pass like this:

lists/tests.py

self.assertTrue(html.strip().endswith('</html>'))

It’s a tiny bit of a cheat, but whitespace at the end of an HTML file really shouldn’t matter to us. Let’s try running the tests again:

$ **python manage.py test**

[...]

OK

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.

**The Django Test Client**

One way we could test this is to manually render the template ourselves in the test, and then compare that to what the view returns. Django has a function called render\_to\_string which will let us do that:

lists/tests.py

from django.template.loader import render\_to\_string

[...]

def test\_home\_page\_returns\_correct\_html(self):

request = HttpRequest()

response = home\_page(request)

html = response.content.decode('utf8')

expected\_html = render\_to\_string('home.html')

self.assertEqual(html, expected\_html)

But that’s a bit of an unwieldy way of testing that we use the right template. And all this faffing about with .decode() and .strip() is distracting. Instead, Django gives us a tool called the [Django Test Client](https://docs.djangoproject.com/en/1.11/topics/testing/tools/#the-test-client), which has built-in ways of checking what templates are used. Here’s how it looks:

lists/tests.py

def test\_home\_page\_returns\_correct\_html(self):

response = self.client.get('/')

html = response.content.decode('utf8')

self.assertTrue(html.startswith('<html>'))

self.assertIn('<title>To-Do lists</title>', html)

self.assertTrue(html.strip().endswith('</html>'))

self.assertTemplateUsed(response, 'home.html')

|  |  |
| --- | --- |
|  | Instead of manually creating an HttpRequest object and calling the view function directly, we call self.client.get, passing it the URL we want to test. |
|  | We’ll leave the old tests there for now, just to make sure everything is working the way we think it is. |
|  | .assertTemplateUsed is the test method that the Django TestCase class provides us. It 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 2 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

self.assertTemplateUsed(response, 'wrong.html')

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. While we’re at it, we can delete our old assertions. And we can also delete the old test\_root\_​url\_resolves test, because that’s tested implicitly by the Django Test Client. We’ve combined two long-winded tests into one!

lists/tests.py (ch04l010)

from django.test import TestCase

class HomePageTest(TestCase):

def test\_uses\_home\_template(self):

response = self.client.get('/')

self.assertTemplateUsed(response, 'home.html')

The main point, though, is that instead of testing constants we’re testing our implementation. Great![[3](http://www.obeythetestinggoat.com/book/chapter_philosophy_and_refactoring.html#_footnote_3)]

Why Didn’t We Just Use the Django Test Client All Along?

You may be asking yourself, "Why didn’t we just use the Django Test Client from the very beginning?" In real life, that’s what I would do. But I wanted to show you the "manual" way of doing it first for a couple of reasons. Firstly because it allowed me to introduce concepts one by one, and keep the learning curve as shallow as possible. Secondly, because you may not always be using Django to build your apps, and testing tools may not always be available—​but calling functions directly and examining their responses is always possible!

The Django Test Client does also have disadvantages; [later in the book](http://www.obeythetestinggoat.com/book/chapter_purist_unit_tests.html) we’ll discuss the difference between fully isolated unit tests and the "integrated" tests that the test client pushes us towards. But for now, it’s very much the pragmatic choice.

**On Refactoring**

That was an absolutely trivial example of refactoring. But, as Kent Beck puts it in [*Test-Driven Development: By Example*](http://www.obeythetestinggoat.com/book/bibliography.html#tddbe), "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; delete the three superfluous assertions, leaving just a check of the contents 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 couple of 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 any more. If you don’t want to end up like [Refactoring Cat](http://bit.ly/1iXyRt4) ([Refactoring Cat—​be sure to look up the full animated GIF (source: 4GIFs.com)](http://www.obeythetestinggoat.com/book/chapter_philosophy_and_refactoring.html#RefactoringCat)), stick to small steps; keep refactoring and functionality changes entirely separate.

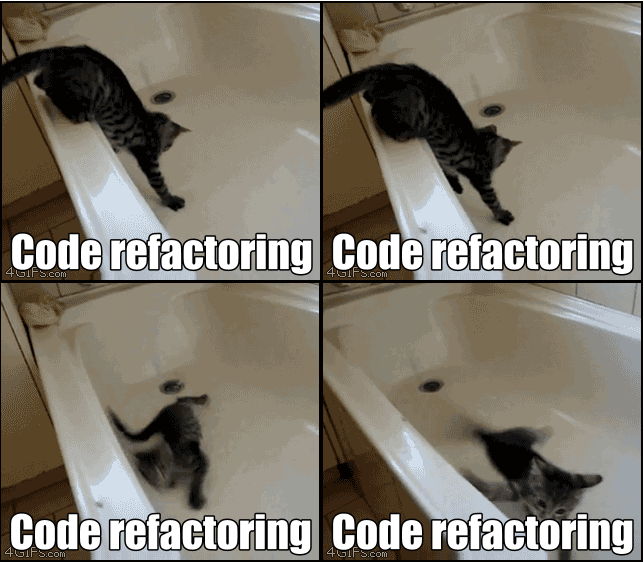


Figure 2. Refactoring Cat—​be sure to look up the full animated GIF (source: 4GIFs.com)

|  |  |
| --- | --- |
|  | We’ll come across “Refactoring Cat” again during this book, as an example of what happens when we get carried away and want to 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 home page view to use a template"**

**A Little More of Our Front Page**

In the meantime, our functional test 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. We wanted an <h1>:

lists/templates/home.html

<html>

<head>

<title>To-Do lists</title>

</head>

<body>

<h1>Your To-Do list</h1>

</body>

</html>

Let’s see if our functional test likes it a little better:

selenium.common.exceptions.NoSuchElementException: Message: Unable to locate

element: [id="id\_new\_item"]

OK…​

lists/templates/home.html

[...]

<h1>Your To-Do list</h1>

<input id="id\_new\_item" />

</body>

[...]

And now?

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

We add our placeholder text…​

lists/templates/home.html

<input id="id\_new\_item" placeholder="Enter a to-do item" />

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

<input id="id\_new\_item" placeholder="Enter a to-do item" />

<table id="id\_list\_table">

</table>

</body>

Now what does the FT say?

File "functional\_tests.py", line 43, in

test\_can\_start\_a\_list\_and\_retrieve\_it\_later

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

self.assertTrue(

any(row.text == '1: Buy peacock feathers' for row in rows),

"New to-do item did not appear in table"

)

If you run the FT again, you should see our 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, years misspent as a management consultant have ruined me. On the plus side, it will feature recursion.

What is the overall TDD process? See [Overall TDD process](http://www.obeythetestinggoat.com/book/chapter_philosophy_and_refactoring.html#simple-TDD-diagram).

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. Then, optionally, we might refactor our code, using our tests to make sure we don’t break anything.

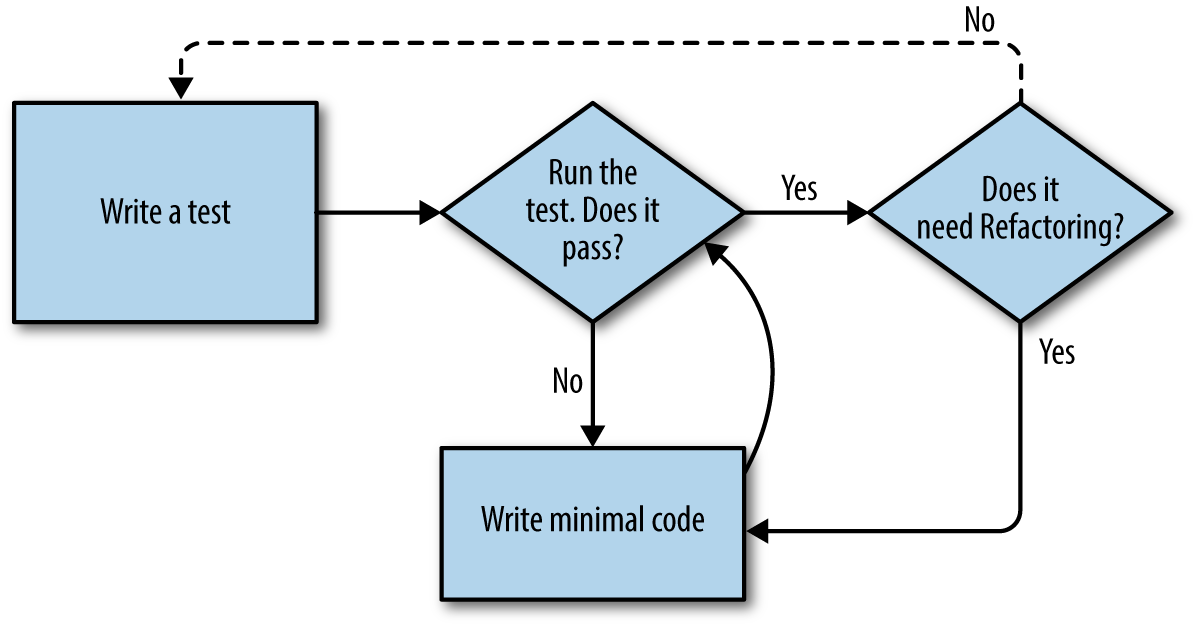


Figure 3. Overall TDD process

But how does this apply when we have functional tests *and* unit tests? Well, you can think of the functional test as being a high-level view of the cycle, where "writing the code" to get the functional tests to pass actually involves using another, smaller TDD cycle which uses unit tests. See [The TDD process with functional and unit tests](http://www.obeythetestinggoat.com/book/chapter_philosophy_and_refactoring.html#Double-Loop-TDD-diagram).

We write a functional test and see it fail. Then, the process of "writing code" to get it to pass is a mini-TDD cycle of its own: we write one or more unit tests, and go into the unit-test/code cycle until the unit tests pass. Then, we go back to our FT to check that it gets a little further, and we can write a bit more of our application—​using more unit tests, and so on.

What about refactoring, in the context of functional tests? Well, that means we use the functional test to check that we’ve preserved the behaviour of our application, but we can change or add and remove unit tests, and use a unit test cycle to actually change the implementation.

The functional tests are the ultimate judge of whether your application works or not. The unit tests are a tool to help you along the way.

This way of looking at things is sometimes called "Double-Loop TDD". One of my eminent tech reviewers, Emily Bache, wrote [a blog post](http://bit.ly/1iXzoLR) on the topic, which I recommend for a different perspective.

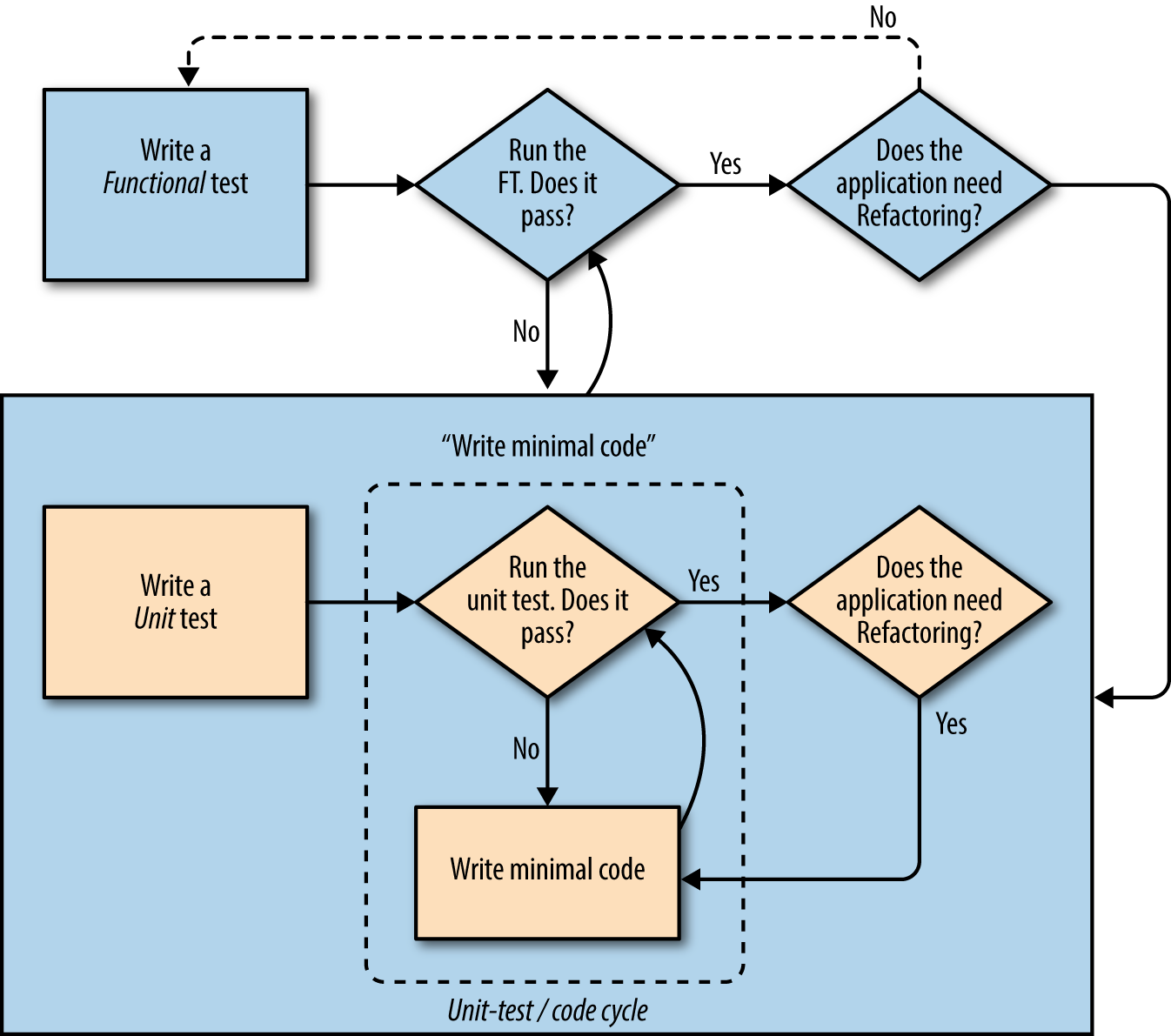


Figure 4. The TDD process with functional and unit tests

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](https://github.com/hjwp/book-example/) on GitHub. 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 is [here](https://github.com/hjwp/book-example/tree/chapter_philosophy_and_refactoring), for example. It 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]](http://www.obeythetestinggoat.com/book/appendix_github_links.html), as well as instructions on how to download them or use Git to compare your code to mine.

[1](http://www.obeythetestinggoat.com/book/chapter_philosophy_and_refactoring.html#_footnoteref_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](http://www.obeythetestinggoat.com/book/chapter_philosophy_and_refactoring.html#_footnoteref_2). 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](http://bit.ly/1iXxWZL).

[3](http://www.obeythetestinggoat.com/book/chapter_philosophy_and_refactoring.html#_footnoteref_3). Are you unable to move on because you’re wondering what those *ch04l0xx* things are, next to some of the code listings? They refer to specific [commits](https://github.com/hjwp/book-example/commits/chapter_philosophy_and_refactoring) 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/master/tests). You know, the tests for the tests in the book about testing. They have tests of their own, naturally.

Last updated 2017-11-21 06:25:12 GMT