**Test-Driven Development with Python**

Harry Percival

Copyright © 2014 Harry Percival

Free Online:

<http://chimera.labs.oreilly.com/books/1234000000754/index.html>

Code examples are available at <https://github.com/hjwp/book-example/>; you’ll find branches for each chapter there (eg,[https://github.com/hjwp/book-example/tree/chapter\_03](https://github.com/hjwp/book-example/tree/chapter_03" \t "_top)).

“Test-Driven Development with Python by Harry Percival (O’Reilly). Copyright 2014 Harry Percival, 978-1-449-36482-3.”

**PythonAnywhere**: Host, run and code Python in the Cloud!

<https://www.pythonanywhere.com/>

Django, Flask, SqlAlchemy etc.

Postgres (paid), MySQL and SqlLite (free)

**HTML tutorials:** <http://www.webplatform.org/>

**Also Need:**

Firefox browser

Git

Pip

**Using PythonAnywhere**

**Chapter 1. Get Django Setup Using a Functional Test**

Use the PythonAnywhere quick-start option in the Web tab. Add a new web app, choose Django, Python 3, and then use superlists as the project name.

# alternatively: **django-admin.py startproject superlists**

.

├── functional\_tests.py

└── superlists

├── manage.py

└── superlists

├── \_\_init\_\_.py

├── settings.py

├── urls.py

└── wsgi.py

/home/jizhenzhao/superlists/superlists/

Stores stuff for the whole project, e.g.

settings.py

/home/jizhenzhao/superlists/manage.py m

Manage the application e.g. run the development serveranage.py

**python3 manage.py runserver**

**functional\_tests.py**

from **selenium** import webdriver

browser = webdriver.Firefox()

#browser.get('http://localhost:8000')

browser.get('http://jizhenzhao.pythonanywhere.com')

assert 'Django' **in** browser.title

**$ xvfb-run python3 functional\_tests.py**

There should have no errors.

Xvfb: X Virtual Frame Buffer

**Git setup and initial commit**

02:31 ~/superlists $ mv ../Test-Driven\_Development\_with\_Python/functional\_tests.py .

02:31 ~/superlists $ ls

functional\_tests.py manage.py media static superlists

02:31 ~/superlists $ git init .

Initialized empty Git repository in /home/jizhenzhao/superlists/.git/

02:32 ~/superlists $ ls

functional\_tests.py manage.py media static superlists

02:34 ~/superlists $ echo "db.sqlite3" >> .gitignore

02:35 ~/superlists $ git add .

02:35 ~/superlists $ git status

02:37 ~/superlists $ git add .gitignore

02:37 ~/superlists $ git config --global user.email “[jizhen.zhao@gmail.com](mailto:jizhen.zhao@gmail.com)”

02:41 ~/superlists $ git config --global user.name "jizhen zhao"

02:41 ~/superlists $ git commit

[master (root-commit) 1110cd9] Init commit for superlists project: from the book Test-Driven Development with Python

7 files changed, 127 insertions(+)

create mode 100644 .gitignore

create mode 100644 functional\_tests.py

create mode 100755 manage.py

create mode 100644 superlists/\_\_init\_\_.py

create mode 100644 superlists/settings.py

create mode 100644 superlists/urls.py

create mode 100644 superlists/wsgi.py­

02:44 ~/superlists (master)$`AQW

**Chapter 2. Extending Our Functional Test Using the unittest Module**

1. Using a Functional Test to Scope Out a Minimum Viable App

The following tests: look at how the whole application functions, from the outside.

* Functional Test:

How the application functions from the users’ point of view. Track User Story.

* Acceptance Test
* End-to-End Test
* Black Box Test

1. functional\_tests.py

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*

*# Edith wonders whether the site will remember her list. Then she sees*

*# that the site has generated a unique URL for her -- there is some*

*# explanatory text to that effect.*

*# She visits that URL - her to-do list is still there.*

*# Satisfied, she goes back to sleep*

browser.quit()

1. A Word for Comments…
   1. it’s pointless to write a comment that just repeats what you’re doing with the code, e.g.

*# increment wibble by 1*

wibble += 1

* 1. Not only is it pointless, there’s a danger that you forget to update the comments when you update the code, and they end up being misleading.
  2. But there are other places where comments are very useful.

1. 01:23 ~/superlists (master)**$** xvfb-run python3 functional\_tests.py

Traceback (most recent call last):

File "functional\_tests.py", line 11, in <module>

assert 'To-Do' in browser.titleAssertionError

1. The python statandard library’s unittest module

from **selenium** import webdriver

import **unittest**

class **NewVisitorTest**(unittest.TestCase): *#*1

def **setUp**(self): *#*2

self.browser = webdriver.Firefox()

def **tearDown**(self): *#*3

self.browser.quit()

def **test\_can\_start\_a\_list\_and\_retrieve\_it\_later**(self): *#*4

*# 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) *#*5

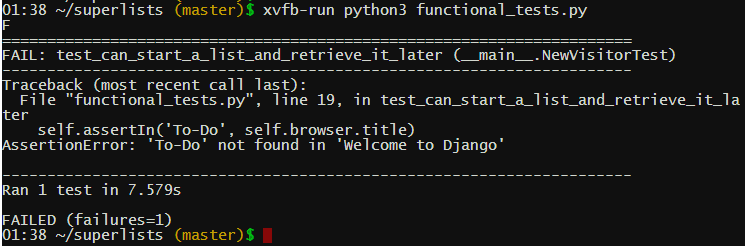
self.fail('Finish the test!') *#*6

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

[...rest of comments as before]

if \_\_name\_\_ == '\_\_main\_\_': *#*7

unittest.main(warnings='ignore') *#*8



1. Implicit waits

[...]

def **setUp**(self):

self.browser = webdriver.Firefox()

self.browser.implicitly\_wait(3)

def **tearDown**(self):

[...]

Be careful! implicitly\_wait won’t work for every use case.

1. Commit

01:42 ~/superlists (master)**$** git diff

01:45 ~/superlists (master)**$** git commit –a

[master 6e7de2e] dded unit test

1 file changed, 46 insertions(+), 6 deletions(-)

rewrite functional\_tests.py (82%)

1. 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.

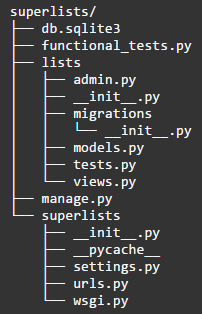
Expected failure

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

**Chapter 3. Testing a Simple Home Page with Unit Tests**

1. Our first Django App, and Our First Unit Test

01:45 ~/superlists (master)**$** python3 manage.py startapp lists



1. Unit Tests, and How They Differ from Functional Tests

The basic distinction, though, is that functional tests test the application from the outside, from the point of view of the user. Unit tests test the application from the inside, from the point of view of the programmer.

The TDD approach I’m following wants our application to be covered by both types of test. Our workflow will look a bit like this:

1. We start by writing a *functional test*, describing the new functionality from the user’s point of view.
2. Once we have a functional test that fails, we start to think about how to write code that can get it to pass (or at least to get past its current failure). We now use one or more *unit tests* to define how we want our code to behave—the idea is that each line of production code we write should be tested by (at least) one of our unit tests.
3. Once we have a failing unit test, we write the smallest amount of *application code* we can, just enough to get the unit test to pass. We may iterate between steps 2 and 3 a few times, until we think the functional test will get a little further.
4. Now we can rerun our functional tests and see if they pass, or get a little further. That may prompt us to write some new unit tests, and some new code, and so on.

You can see that, all the way through, the functional tests are driving what development we do from a high level, while the unit tests drive what we do at a low level.

Does that seem slightly redundant? Sometimes it can feel that way, but functional tests and unit tests do really have very different objectives, and they will usually end up looking quite different.

Functional tests should help you build an application with the right functionality, and guarantee you never accidentally break it. Unit tests should help you to write code that’s clean and bug free.

1. Unit Testing in Django

list/test.py

1. Django’s MVC, URLs, and View Functions

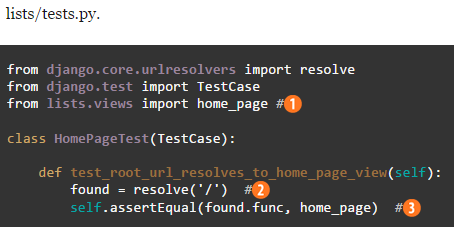
Django is broadly structured along a classic Model-View-Controller (MVC) pattern. Well, broadly. It definitely does have models, but its views are more like a controller, and it’s the templates that are actually the view part, but the general idea is there.

Irrespective of any of that, like any web server, Django’s main job is to decide what to do when a user asks for a particular URL on our site. Django’s workflow goes something like this:

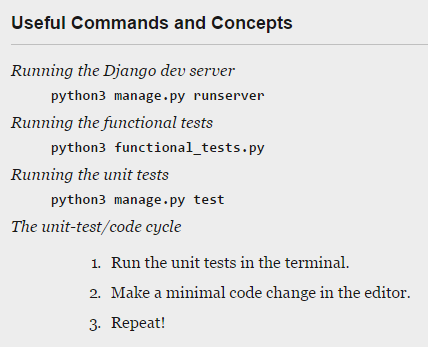
1. An HTTP *request* comes in for a particular *URL*.
2. Django uses some rules to decide which *view* function should deal with the request (this is referred to as *resolving* the URL).
3. The view function processes the request and returns an HTTP *response*.

So we want to test two things:

* Can we resolve the URL for the root of the site (“/”) to a particular view function we’ve made?
* Can we make this view function return some HTML which will get the functional test to pass?



lists/views.py



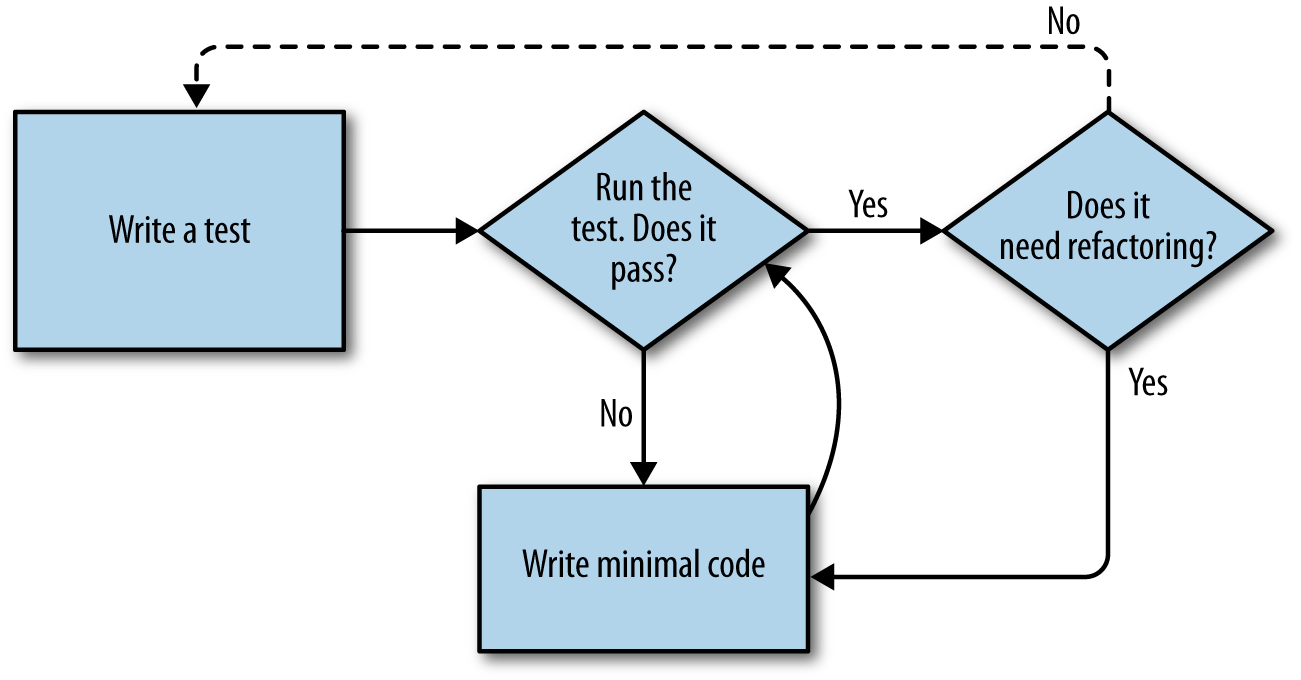
**Chapter 4. What are we doing with all the Tests?**

## Recap: The TDD Process

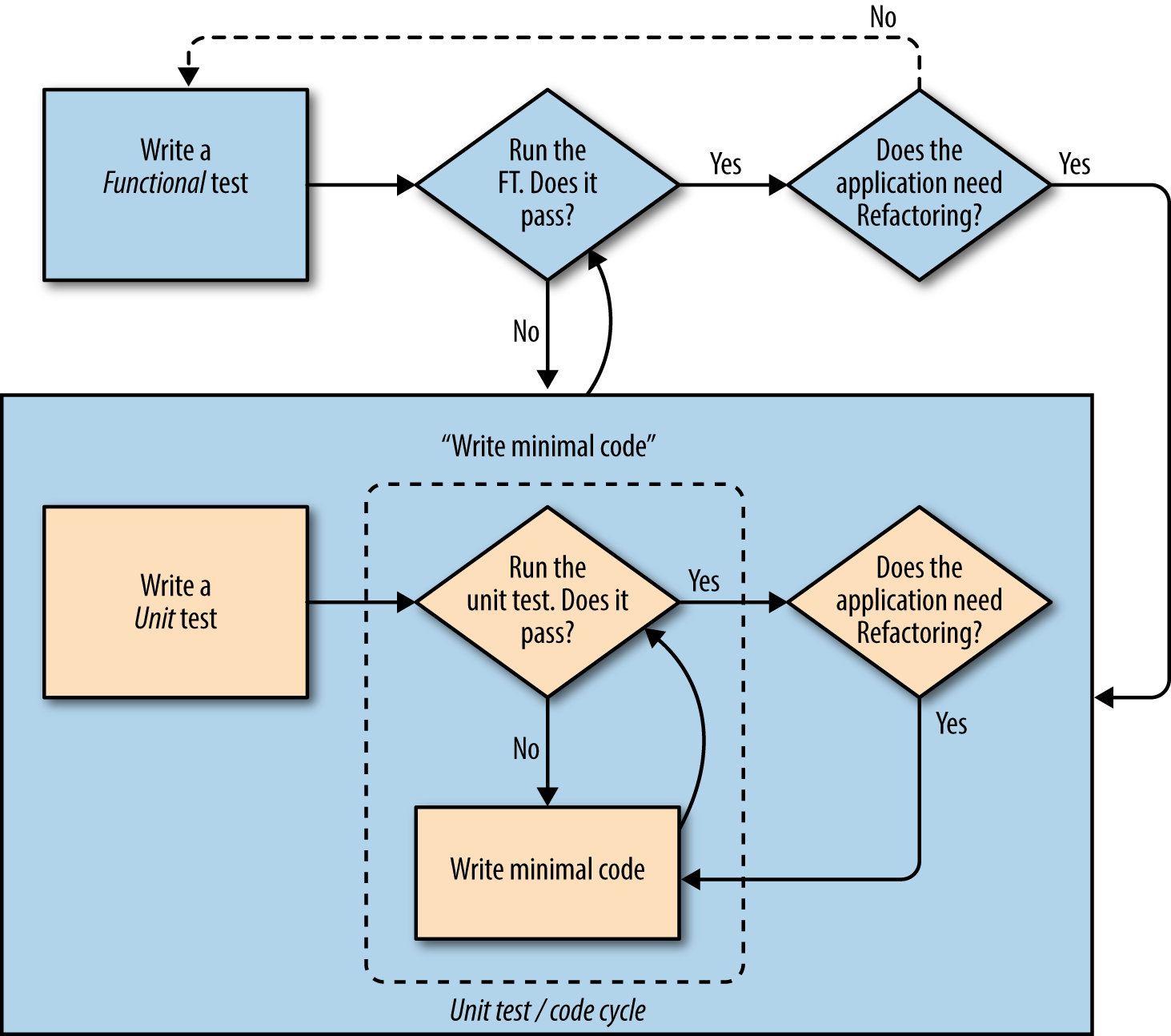
We’ve now seen all the main aspects of the TDD process, in practice:

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

The overall TDD process



*Figure 4-3. Overall TDD process*



*Figure 4-4. The TDD process with functional and unit tests*

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".

**Using Git to check your progress**

If you feel like developing your Git-Fu a little further, you can add my repo as a remote:

git remote add harry https://github.com/hjwp/book-example.git

git fetch harry

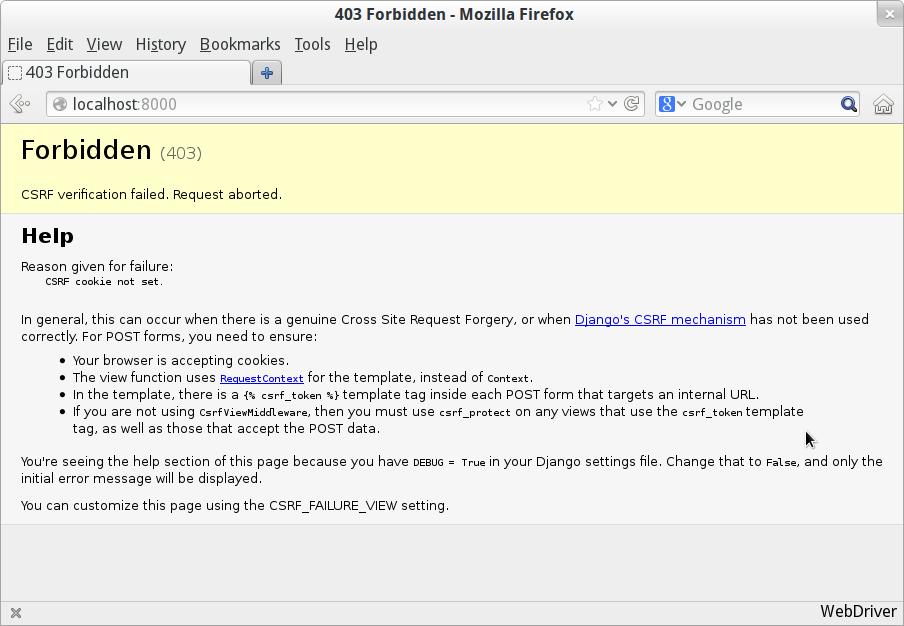
And then, to check your difference from the end of [Chapter 4](http://chimera.labs.oreilly.com/books/1234000000754/ch04.html):

git diff harry/chapter\_04

Git can handle multiple remotes, so you can still do this even if you’re already pushing your code up to GitHub or Bitbucket.

Be aware that the precise order of, say, methods in a class may differ between your version and mine. It may make diffs hard to read.

**Chapter 5. Saving User Inputs**



*Figure 5-1. Django DEBUG page showing CSRF error*

Django’s CSRF protection involves placing a little auto-generated token into each generated form, to be able to identify POST requests as having come from the original site. So far our template has been pure HTML, and in this step we make the first use of Django’s template magic. To add the CSRF token we use a template tag, which has the curly-bracket/percent syntax, {% … %}—famous for being the world’s most annoying two-key touch-typing combination:

lists/templates/home.html.

**<form** method="POST"**>**

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

{% csrf\_token %}

**</form>**

Django will substitute that during rendering with an <input type="hidden"> containing the CSRF token. Rerunning the functional test will now give us an expected failure:

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

**Chapter 1. Get Django Setup Using a Functional Test**

Use the PythonAnywhere quick-start option in the Web tab. Add a new web

03:32 ~ **$** mkvirtualenv django18 --python=/usr/bin/python3.4

Reactivate Virtual Env later: workon django18

me@mymachine:~$ workon env1

(env1)me@mymachine:~$ workon env2

(env2)me@mymachine:~$ workon env1

(env1)me@mymachine:~$ deactivate

me@mymachine:~$

(django18)03:33 ~ **$** pip install -U pip

Successfully installed pip-7.0.1

(django18)03:34 ~ **$** pip install Django

Successfully installed django-1.8.2

(django18)03:43 ~ **$** python -c "import django; print(django.get\_version())"

1.8.2

(django18)12:01 ~ **$** django-admin startproject superlists

Uploads Chapter 5 code