
Test-Driven Development with Python

Test-Driven Development with Python

by

Revision History for the :

See <http://oreilly.com/catalog/errata.csp?isbn=> for release details.

Table of Contents

Preface	vii
1. Getting Django set up using a Functional Test	1
Obey the Testing Goat: Do nothing until you have a test	3
Getting Django up and running	3
Optional: Starting a Git repository	5
2. Extending our Functional Test using the unittest module	7
Using the Functional Test to scope out a minimum viable app	7
The Python standard library's <code>unittest</code> module	7
Optional: Commit	9
3. Testing a simple home page with unit tests	11
Our first Django app, and our first unit test	11
Unit tests, and how they differ from Functional tests	11
Unit testing in Django	12
Django's MVC, URLs and view functions	15
Unit testing a view	15
The unit test / code cycle	15
Unit testing URL mapping	18
urls.py	18
4. What are we doing with all these tests?	21
A moment's reflection - what are we up to?	21
Using Selenium to test user interactions	22
The "Don't test constants" rule, and templates to the rescue	23
Refactoring	24
On refactoring	25
A little more of our front page	26

Recap: the TDD process	27
5. Saving user input	31
Wiring up our form to send a POST request	31
Processing a POST request on the server	33
Template context	33
3 strikes and refactor	35
The Django ORM & our first model	36
Saving the POST to the database	37
Redirect after a POST	39
Rendering items in the template	40
Creating our production database with syncdb	41
6. Getting to the minimum viable site	45
Ensuring test isolation in functional tests	45
Small Design When Necessary	48
REST	48
Implementing the new design using TDD	49
Iterating towards the new design	50
Testing views, templates and URLs together with the Django Test Client	50
Adding another URL	54
Adjusting our models	56
The final stage: each list should have its own URL	59
A final refactor using URL includes	63
7. Outline to date & future chapters plan	65
BOOK 1: Building a minimum viable app with TDD	65
BOOK 2: Growing the site	65
Chapter 9: User Authentication + the admin site	65
Chapter 10: A more complex model, forms and validation	66
Chapter 11: javascript	66
Chapter 11: Ajax	66
Chapter 12: sharing lists	66
Chapter 13: oauth	66
More/Other possible contents	66
BOOK 3: Trendy stuff	67
Chapter 14: CI	67
Chapter 15 & 16: More Javascript	67
Chapter 17: Async	68
Chapter 18: NoSQL	68
Chapter 19: Caching	68
Appendices	68

Other possible appendix(?) topics	68
Existing appendix I: PythonAnywhere	69

Preface

This handout will contain code listings and expected test run output, for you to copy out and check against. I'll demo things first each time. It's a massively stripped down version of my book, more info at

<http://www.obeythetestinggoat.com>

Outline:

- Chapter 1 — Use the simplest possible selenium FT to install Django
- Chapter 2 — Switch to unittest
- Chapter 3 — Unit tests for a view and a URL mapping
- Chapter 4 — Refactoring: switch to using templates
- Chapter 5 — A first attempt at saving POST requests to the database
- Chapter 6 — Step-by-step changes to get to a better solution

Pre-requisites

System software:

- **Firefox**
- **Git**
- **pip** (google “Python pip”)

Python modules:

- **Django** (`pip install --upgrade django`). (need v 1.5)

- **Selenium** (`pip install --upgrade selenium`), (absolute latest possible)

Cloning the repo

Cloning my repo — optional, but will occasionally save you a bit of typing

```
git clone https://github.com/hjwp/book-example.git tdd-workshop
cd tdd-workshop/
git checkout -b fresh-start
git rm -r *
git commit -m"delete everything to start from scratch"
```

If you're an experienced github user, feel free to fork my repo instead.

I show you how to do git commits at each stage. these are optional, but they can help you to check your progress against what's in my repo, eg

```
git diff origin/chapter_4
```

would show you the diff between your code and mine, where the `chapter_4` branch is the code at the *end* of chapter 4.

Getting Django set up using a Functional Test

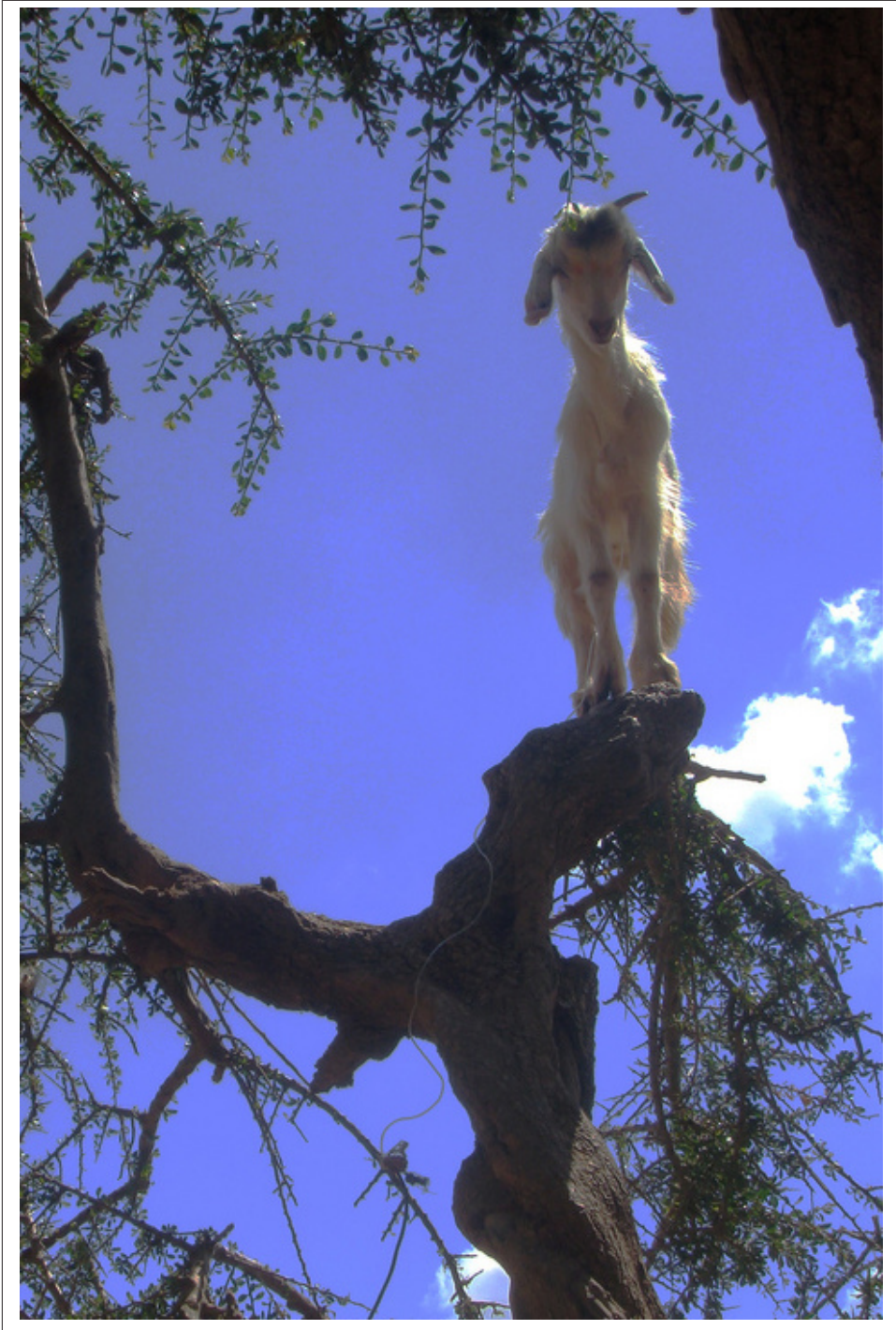


Figure 1-1. Goats are more agile than you think (image credit: *Caitlin Stewart, on*

Flickr)

Obey the Testing Goat: Do nothing until you have a test

Inside the *tdd-workshop* folder

```
from selenium import webdriver  
  
browser = webdriver.Firefox()  
browser.get('http://localhost:8000')  
  
assert 'Django' in browser.title
```

functional_tests.py.

Running it:

```
$ python functional_tests.py  
Traceback (most recent call last):  
  File "functional_tests.py", line 6, in <module>  
    assert 'Django' in browser.title  
AssertionError
```

Getting Django up and running

```
$ django-admin.py startproject superlists .
```



Don't miss the `.` at the end — it's important

That will create a tree like this:

```
tdd-workshop  
├── functional_tests.py  
├── manage.py  
└── superlists  
    ├── __init__.py  
    ├── settings.py  
    ├── urls.py  
    └── wsgi.py
```



If you have two folders called *superlists*, you've done it wrong. Start again!

```
$ python manage.py runserver  
Validating models...
```

```
0 errors found
Django version 1.5.1, using settings 'superlists.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Leave that running, and open another command shell. In that, we can try running our test again (from the folder we started in):

```
$ python functional_tests.py
$
```

Not much action on the command-line, but you should notice two things: Firstly, there was no ugly `AssertionError` and secondly, the Firefox window that Selenium pops up has a different-looking page on it.

Well, it may not look like much, but that was our first ever passing test! Hooray!

If it all feels a bit too much like magic, like it wasn't quite real, why not go and take a look at the dev server manually, by opening a web browser yourself and visiting <http://localhost:8000>. You should see something like [Figure 1-2](#)

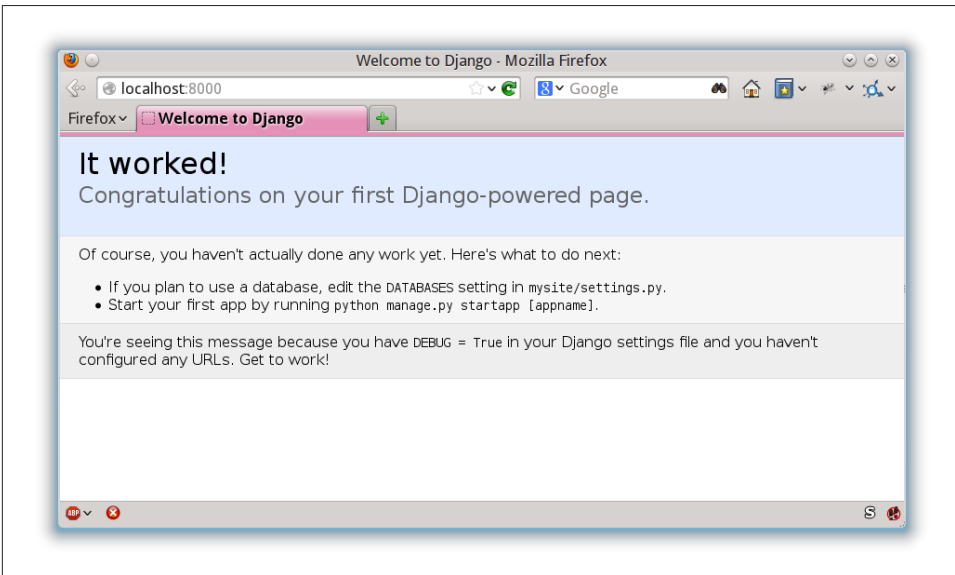


Figure 1-2. It Worked!

You can quit the development server now if you like, back in the original shell, using `Ctrl+C`.

Optional: Starting a Git repository

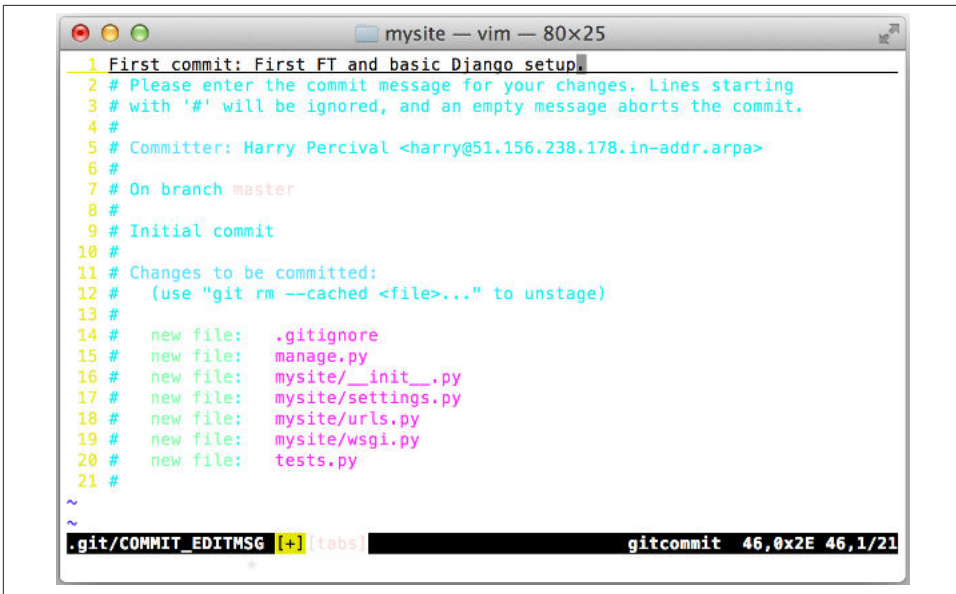
Let's start by moving *functional_tests.py* into the *superlists* folder, and doing the `git init` to start the repository:



from this point onwards, the top-level *superlists* folder will be our working directory. Whenever I show a command to type in, it will assume we're in this directory. Similarly, if I mention a path to a file, it will be relative to this top-level directory. So *superlists/settings.py* means the *settings.py* inside the second-level *superlists*. Clear as mud? If in doubt, look for *manage.py* — you want to be in the same directory as *manage.py*.

```
$ ls
functional_tests.py manage.py  superlists
# ignore .pyc files:
$ echo ".pyc" > .gitignore
$ *git add *
$ git status
# On branch fresh-start
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   .gitignore
#       new file:   functional_tests.py
#       new file:   manage.py
#       new file:   superlists/__init__.py
#       new file:   superlists/settings.py
#       new file:   superlists/urls.py
#       new file:   superlists/wsgi.py
#
$ git commit
```

When you type `git commit`, it will pop up an editor window for you to write your commit message in. Mine looked like [Figure 1-3](#):



```
1 First commit: First FT and basic Django setup
2 # Please enter the commit message for your changes. Lines starting
3 # with '#' will be ignored, and an empty message aborts the commit.
4 #
5 # Committer: Harry Percival <harry@51.156.238.178.in-addr.arpa>
6 #
7 # On branch master
8 #
9 # Initial commit
10 #
11 # Changes to be committed:
12 #   (use "git rm --cached <file>..." to unstage)
13 #
14 #   new file:   .gitignore
15 #   new file:   manage.py
16 #   new file:   mysite/__init__.py
17 #   new file:   mysite/settings.py
18 #   new file:   mysite/urls.py
19 #   new file:   mysite/wsgi.py
20 #   new file:   tests.py
21 #
~
.git/COMMIT_EDITMSG [1] | tabs | gitcommit 46,0x2E 46,1/21
```

Figure 1-3. First Git Commit

Congratulations! You've written a functional test using Selenium, and you've got Django installed and running, in a certifiable, test-first, goat-approved TDD way.

Extending our Functional Test using the unittest module

Using the Functional Test to scope out a minimum viable app

- Functional tests, aka acceptance tests, black-box tests
- User story as comments
- switch to unittest

The Python standard library's `unittest` module

functional_tests.py.

```
import unittest
from selenium import webdriver

class NewVisitorTest(unittest.TestCase): #1

    def setUp(self): #2
        self.browser = webdriver.Firefox()
        self.browser.implicitly_wait(3) #3

    def tearDown(self): #4
        self.browser.quit()

    def test_can_start_a_list_and_retrieve_it_later(self): #5
        # 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) #6
self.fail('Finish the test!') #7

# 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

if __name__ == '__main__': #8
    unittest.main()

```

You can copy & paste this if you like, or use

```
git checkout origin/chapter_2 -- functional_tests.py
```



make sure I explain each of the numbers!

```

$ 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 18, in
test_can_start_a_list_and_retrieve_it_later
    self.assertIn('To-Do', self.browser.title)
AssertionError: 'To-Do' not found in u'Welcome to Django'

-----

Ran 1 test in 1.747s

FAILED (failures=1)

```


Optional: Commit

Do a **git status** — that should assure you that the only file that has changed is *functional_tests.py*. Then do a **git diff**, which shows you the difference between the last commit and what’s currently on disk. That should tell you that *functional_tests.py* has changed quite substantially:

```
$ git diff
diff --git a/functional_tests.py b/functional_tests.py
index d333591..b0f22dc 100644
--- a/functional_tests.py
+++ b/functional_tests.py
@@ -1,6 +1,45 @@
+import unittest
+from selenium import webdriver

-browser = webdriver.Firefox()
-browser.get('http://localhost:8000')
+class NewVisitorTest(unittest.TestCase):

-assert 'Django' in browser.title
+    def setUp(self):
+        self.browser = webdriver.Firefox()
+        self.browser.implicitly_wait(3)
+
+    def tearDown(self):
+        self.browser.quit()
[...]
```

Now let’s do a:

```
$ git commit -a
```

The **-a** means “automatically add any changes to tracked files”

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

Now we’re in an excellent position to start writing some real code for our lists app.

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 a way that we expected it to

Testing a simple home page with unit tests

Our first Django app, and our first unit test

Projects are made up of “apps”...

```
$ python manage.py startapp lists
```

That will create a folder at *superlists/lists*, next to *superlists/superlists*, and within it a number of placeholder files for models, views and, of immediate interest to us, tests.

```
tdd-workshop
├── functional_tests.py
├── lists
│   ├── __init__.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
├── manage.py
└── superlists
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

Unit tests, and how they differ from Functional tests



a good place to ask questions, if I don't explain myself well here!

Unit testing in Django

Open up `lists/tests.py` and you'll see something like this:

```
"""
    This file demonstrates writing tests using the unittest module. These will pass
    when you run "manage.py test".

    Replace this with more appropriate tests for your application.
    """
```

lists/tests.py.

```
from django.test import TestCase
```

```
class SimpleTest(TestCase):
    def test_basic_addition(self):
        """
        Tests that 1 + 1 always equals 2.
        """
        self.assertEqual(1 + 1, 2)
```

Let's deliberately break the test and see if we can see it fail.

```
self.assertEqual(1 + 1, 3)
```

lists/tests.py.

Now let's invoke this mysterious Django test runner. As usual, it's a `manage.py` command:

```
$ python manage.py test
```

```
[... lots and lots of traceback]
```

```
Traceback (most recent call last):
```

```
File "/usr/local/lib/python2.7/dist-packages/django/test/testcases.py", line
259, in __call__
    self._pre_setup()
File "/usr/local/lib/python2.7/dist-packages/django/test/testcases.py", line
479, in _pre_setup
    self._fixture_setup()
File "/usr/local/lib/python2.7/dist-packages/django/test/testcases.py", line
829, in _fixture_setup
    if not connections_support_transactions():
File "/usr/local/lib/python2.7/dist-packages/django/test/testcases.py", line
816, in connections_support_transactions
    for conn in connections.all()
File "/usr/local/lib/python2.7/dist-packages/django/test/testcases.py", line
816, in <genexpr>
    for conn in connections.all()
File "/usr/local/lib/python2.7/dist-packages/django/utils/functional.py",
line 43, in __get__
    res = instance.__dict__[self.func.__name__] = self.func(instance)
File "/usr/local/lib/python2.7/dist-packages/django/db/backends/__init__.py",
```

```

line 442, in supports_transactions
    self.connection.enter_transaction_management()
File
"/usr/local/lib/python2.7/dist-packages/django/db/backends/dummy/base.py", line
15, in complain
    raise ImproperlyConfigured("settings.DATABASES is improperly configured. "
ImproperlyConfigured: settings.DATABASES is improperly configured. Please
supply the ENGINE value. Check settings documentation for more details.

```

Ran 85 tests in 0.788s

FAILED (errors=404, skipped=1)
AttributeError: _original_allowed_hosts

Fix in *superlists/settings.py*

```

                                                                 superlists/settings.py.
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.', # Add 'postgresql_psycopg2', 'mysql',
                                         # 'sqlite3' or 'oracle'.
        'NAME': '',                      # Or path to database file if using sqlite3.
        # The following settings are not used with sqlite3:
        'USER': '',
        'PASSWORD': '',
        'HOST': '',                       # Empty for localhost through domain
                                         # sockets or '127.0.0.1' for localhost
                                         # through TCP.
        'PORT': '',                       # Set to empty string for default.
    }
}

```

sqlite3 is the quickest to set up.

```

                                                                 superlists/settings.py.
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': '',                      # Or path to database file if using sqlite3.
    }
}

```

Is that enough? Let's try running the test again:

```

$ python manage.py test
Creating test database for alias 'default'...
.....S.....
.....X.....
.....

```

Ran 479 tests in 17.679s

```
OK (skipped=1, expected failures=1)
Destroying test database for alias 'default'...
```

479 tests! But where is our failure?

```
$ python manage.py test lists
ImproperlyConfigured: App with label lists could not be found
```

BUT IT'S RIGHT THERE!

superlists/settings.py.

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # Uncomment the next line to enable the admin:
    # 'django.contrib.admin',
    # Uncomment the next line to enable admin documentation:
    # 'django.contrib.admindocs',
    'lists',
)
```

Now we can try running the tests for lists again:

```
$ python manage.py test lists
Creating test database for alias 'default'...
F
=====
FAIL: test_basic_addition (lists.tests.SimpleTest)
-----
Traceback (most recent call last):
  File "/workspace/superlists/lists/tests.py", line 16, in test_basic_addition
    self.assertEqual(1 + 1, 3)
AssertionError: 2 != 3
-----

Ran 1 test in 0.000s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

That's more like it! If you like, you can reassure yourself that it gets run as part of the general command, `manage.py test` as well, and you should see it now runs 480 tests instead of 479.

This is a good point for a commit:

```
$ git status
# should show you superlists/settings.py has changed and lists/ is untracked

$ git add superlists/settings.py
```

```
$ git add lists
$ git diff --staged # will show you the diff that you're about to commit
$ git commit -m"Add app for lists, with deliberately failing unit test"
```

As no doubt you've guessed, the `-m` flag lets you pass in a commit message at the command-line, so you don't need to go via an editor. It's up to you to pick the way you like to use the `git` command-line, I'll just show you the main ones I've seen used.

Django's MVC, URLs and view functions

- MVC
- resolving URLs
- view functions

Unit testing a view

lists/tests.py.

```
from django.test import TestCase
from django.http import HttpRequest

from lists.views import home_page

class HomePageTest(TestCase):

    def test_home_page_returns_correct_html(self):
        request = HttpRequest() #1
        response = home_page(request) #2
        self.assertTrue(response.content.startswith('<html>')) #3
        self.assertIn('<title>To-Do lists</title>', response.content) #4
        self.assertTrue(response.content.endswith('</html>')) #5
```



make sure I explain all those numbers!

Let's run the unit tests now and see how we get on:

```
ImportError: cannot import name home_page
```

Obviously.

The unit test / code cycle

We can start to settle into the TDD *unit test / code cycle* now:

- in the terminal, run the unit tests and see how they fail
- in the editor, make a minimal code change to address the current test failure

And repeat!

What's the simplest possible code change that will fix the test failure?

```
# Create your views here.
home_page = None
```

lists/views.py.

Yep, seriously.

```
=====
ERROR: test_home_page_returns_correct_html (lists.tests.HomePageTest)
-----
Traceback (most recent call last):
  File "/tmp/tdd-workshop/lists/tests.py", line 11, in test_home_page_returns_correct_html
    response = home_page(request)
TypeError: 'NoneType' object is not callable
-----
Ran 1 test in 0.001s

FAILED (errors=1)
Destroying test database for alias 'default'...
```

Ok then...

```
def home_page():
    pass
```

lists/views.py.

And now?

```
TypeError: home_page() takes no arguments (1 given)
```

OK then..

```
def home_page(request):
    pass
```

lists/views.py.

- Tests:

```
self.assertTrue(response.content.startswith('<html>'))
AttributeError: 'NoneType' object has no attribute 'content'
```

- Code - we use `django.http.HttpResponse`, as predicted:

```
from django.http import HttpResponse

def home_page(request):
    return HttpResponse()
```

lists/views.py.

- Tests again:

```
self.assertTrue(response.content.startswith('<html>'))
AssertionError: False is not true
```

- Code again:

```
def home_page(request):
    return HttpResponse('<html>')
```

lists/views.py.

- Tests:

```
AssertionError: '<title>To-Do lists</title>' not found in '<html>'
```

- Code:

```
def home_page(request):
    return HttpResponse('<html><title>To-Do lists</title>')
```

lists/views.py.

- Tests — almost there?

```
self.assertTrue(response.content.endswith('</html>'))
AssertionError: False is not true
```

- Come on, one last effort:

```
def home_page(request):
    return HttpResponse('<html><title>To-Do lists</title></html>')
```

lists/views.py.

- Surely?

```
$ python manage.py test lists
Creating test database for alias 'default'...
```

```
·
```

```
-----
Ran 1 test in 0.001s
```

```
OK
```

```
Destroying test database for alias 'default'...
```

YES! Does that get our FT to pass?

No.

```
$ python functional_tests.py
[...]
AssertionError: 'To-Do' not found in u'Welcome to Django'
```

Unit testing URL mapping

Add a new test method:

lists/tests.py.

```
from django.core.urlresolvers import resolve
from django.test import TestCase
from lists.views import home_page

class HomePageTest(TestCase):

    def test_home_page_returns_correct_html(self):
        request = HttpRequest()
        response = home_page(request)
        self.assertTrue(response.content.startswith('<html>'))
        self.assertIn('<title>To-Do lists</title>', response.content)
        self.assertTrue(response.content.endswith('</html>'))

    def test_root_url_resolves_to_home_page_view(self):
        found = resolve('/')
        self.assertEqual(found.func, home_page)
```

```
$ python manage.py test lists
Creating test database for alias 'default'...
E
=====
ERROR: test_root_url_resolves_to_home_page_view (lists.tests.HomePageTest)
-----
Traceback (most recent call last):
  File "/workspace/superlists/lists/tests.py", line 8, in
test_root_url_resolves_to_home_page_view
    found = resolve('/')
  File "/usr/local/lib/python2.7/dist-packages/django/core/urlresolvers.py",
line 440, in resolve
    return get_resolver(urlconf).resolve(path)
  File "/usr/local/lib/python2.7/dist-packages/django/core/urlresolvers.py",
line 334, in resolve
    raise Resolver404({'tried': tried, 'path': new_path})
Resolver404: {'u'path': '', u'tried': []}
-----
Ran 1 test in 0.002s

FAILED (errors=1)
Destroying test database for alias 'default'...
```

urls.py

Default contents:

superlists/urls.py.

```
from django.conf.urls import patterns, include, url
```

```

# Uncomment the next two lines to enable the admin:
# from django.contrib import admin
# admin.autodiscover()

urlpatterns = patterns('',
    # Examples:
    # url(r'^$', 'superlists.views.home', name='home'),
    # url(r'^superlists/', include('superlists.foo.urls')),

    # Uncomment the admin/doc line below to enable admin documentation:
    # url(r'^admin/doc/', include('django.contrib.admindocs.urls')),

    # Uncomment the next line to enable the admin:
    # url(r'^admin/', include(admin.site.urls)),
)

```

Just uncomment one line, to see what happens:

```

urlpatterns = patterns('',
    # Examples:
    url(r'^$', 'superlists.views.home', name='home'),

```

superlists/urls.py.

And run the unit tests again, **python manage.py test lists**:

```

ViewDoesNotExist: Could not import superlists.views.home. Parent module
superlists.views does not exist.

```

```

urlpatterns = patterns('',
    # Examples:
    url(r'^$', 'lists.views.home_page', name='home'),

```

superlists/urls.py.

And the run the tests again:

```
OK
```

Hooray!

What about the FTs? Feels like we're at the end of the race here, surely this is it... could it be...?)

```

$ 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 20, 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)

FAILED? What? Oh, it's an expected fail? Yes? Yes! We have a web page!

Just a little commit to calm down, and reflect on what we've covered

```
$ git diff # should 2 tests in tests.py, the url in urls.py & the view in views.py
$ git commit -am"Basic view now returns minimal HTML"
```

Not bad — we covered:

- Starting a Django app
- The Django unit test runner
- The difference between FTs and unit tests
- Django url resolving and urls.py
- Django view functions, request and response objects
- And returning basic HTML

Useful commands and concepts

Running the Django dev server

```
python manage.py runserver
```

Running the functional tests

```
python functional_tests.py
```

Running the unit tests

```
python manage.py test lists
```

The unit test / code cycle

- Run the unit tests in the terminal
- Make a minimal code change in the editor
- Repeat!

What are we doing with all these tests?

A moment's reflection - what are we up to?



Figure 4-1. Test ALL the things (original illustration credit *Allie Brosh, Hyperbole and a Half*)

- FT / unit test redundancy?
- Bucket from a well
- TDD as a discipline — kata
- Even ridiculously small/simple tests...

Using Selenium to test user interactions

Where were we at the end of the last chapter? Let's re-run 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.

functional_tests.py.

```
import unittest
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

class NewVisitorTest(unittest.TestCase):

    def setUp(self):
        self.browser = webdriver.Firefox()
        self.browser.implicitly_wait(3)

    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)

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
[...]

```

Let's see how it gets on

```

$ python functional_tests.py
[...]
NoSuchElementException: Message: u'Unable to locate element: {"method":"tag
name","selector":"h1"}' ; Stacktrace: [...]

```

Big changes to a functional test are usually a good thing to commit on their own

```

$ 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

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

```

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*.

Refactoring

Always start by running the tests first:

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

Great! Let's start by taking our HTML string and putting it into its own file. We'll 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:

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

lists/templates/home.html.

Mmmh, syntax-highlighted... Much nicer! Now to change our view function:

```
from django.shortcuts import render
```

lists/views.py.

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

That's a change to the code - do the tests still pass?

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

```
self.assertTrue(response.content.endswith('</html>'))
AssertionError: False is not true
```



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.

Darn, not quite. 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(response.content)` to debug this

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

lists/tests.py.

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


Now we can change the tests, in two steps, running them in between:

```
from django.template.loader import render_to_string
[...]
def test_home_page_returns_correct_html(self):
    request = HttpRequest()
    response = home_page(request)
    self.assertTrue(response.content.startswith('<html>'))
    self.assertIn('<title>To-Do lists</title>', response.content)
    self.assertTrue(response.content.strip().endswith('</html>'))

    expected_html = render_to_string('home.html')
    self.assertEqual(response.content, expected_html)
```

lists/tests.py.

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

```
from django.template.loader import render_to_string
[...]
def test_home_page_returns_correct_html(self):
    request = HttpRequest()
    response = home_page(request)
    expected_html = render_to_string('home.html')
    self.assertEqual(response.content, expected_html)
```

lists/tests.py.



Django has a Test Client with tools for testing templates, which we'll use in later chapters. For now we'll use the low-level tools to make sure we're comfortable with how everything works. No magic!

On refactoring

Am I recommending that you actually work this way? No. I'm recommending that you be *able* to work this way.

— Kent Beck
TDD by example

- Google “Refactoring Cat”

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

```
$ git status # see changes to lists.py, tests.py, views.py + new templates folder
$ git add . # will 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.

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:

```
NoSuchElementException: Message: 'Unable to locate element:
{"method":"id","selector":"id_new_item"}'; Stacktrace: [...]
```

OK...

lists/templates/home.html.

```
<html>
  <head>
    <title>To-Do lists</title>
  </head>
  <body>
    <h1>Your To-Do list</h1>
    <input id="id_new_item" />
  </body>
</html>
```

And now?

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

Let's add our placeholder text...

lists/templates/home.html.

```
<input id="id_new_item" placeholder="Enter a to-do item" />
NoSuchElementException: Message: 'Unable to locate element:
{"method":"id","selector":"id_list_table"}' [...]
```

Let's 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 42, 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*:

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

functional_tests.py

If you run the FT again, you should see our message.

```
AssertionError: 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"Add inputbox and empty list table"
```

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

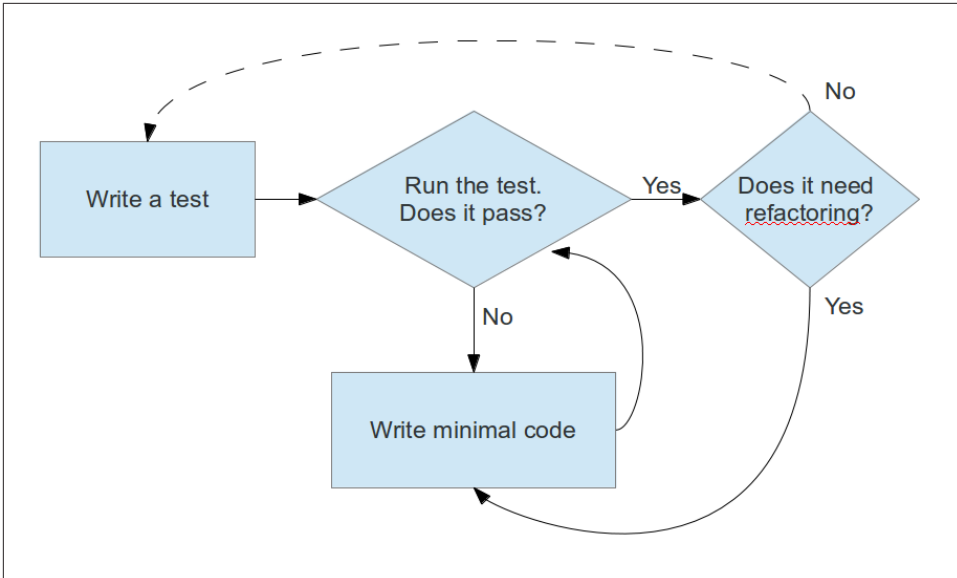


Figure 4-2. Overall TDD process

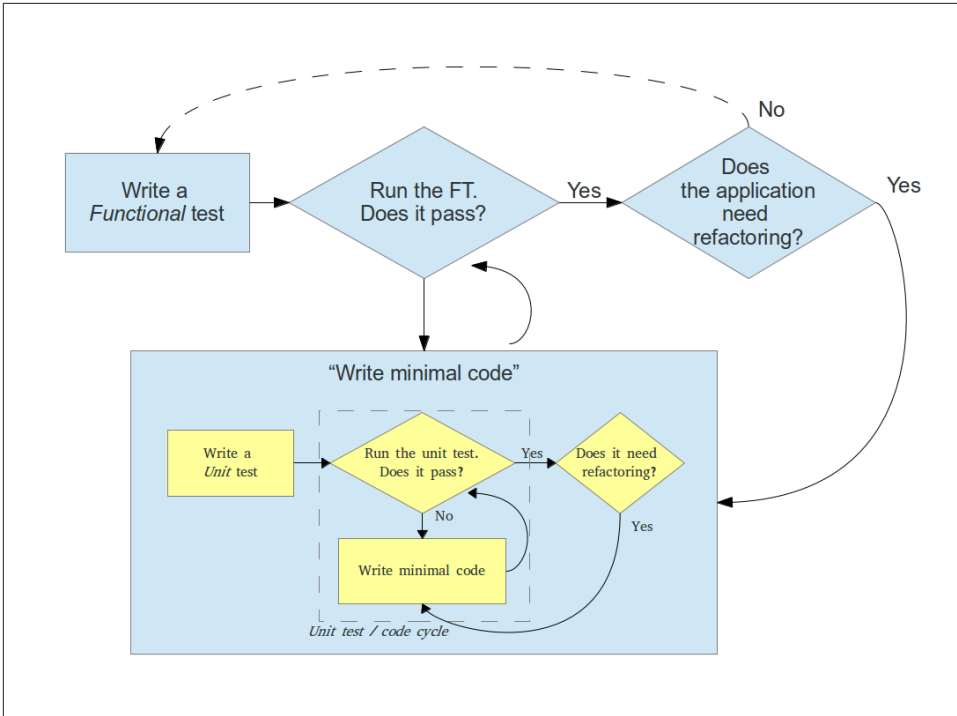


Figure 4-3. The TDD process with Functional and Unit tests

Saving user input

- i. In which I deliberately go wrong, so that TDD can put me right

Wiring up our form to send a POST request

lists/templates/home.html.

```
<h1>Your To-Do list</h1>
<form method="POST">
  <input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />
</form>

<table id="id_list_table">
```

Now, running our FTs gives us a slightly cryptic, unexpected error:

```
$ python functional_tests.py
[...]
Traceback (most recent call last):
  File "functional_tests.py", line 38, in test_can_start_a_list_and_retrieve_it_later
    table = self.browser.find_element_by_id('id_list_table')
[...]
NoSuchElementException: Message: u'Unable to locate element:
{"method":"id","selector":"id_list_table"}' ; Stacktrace [...]
```

When a functional test fails with an unexpected failure, there are several things we can do to debug them:

- Add `print` statements, to show, eg, what the current page text is
- Improve the *error message* to show more info about the current state
- Manually visit the site yourself
- Use `time.sleep` to pause the test during execution

```

# 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)

import time
time.sleep(10)
table = self.browser.find_element_by_id('id_list_table')

```

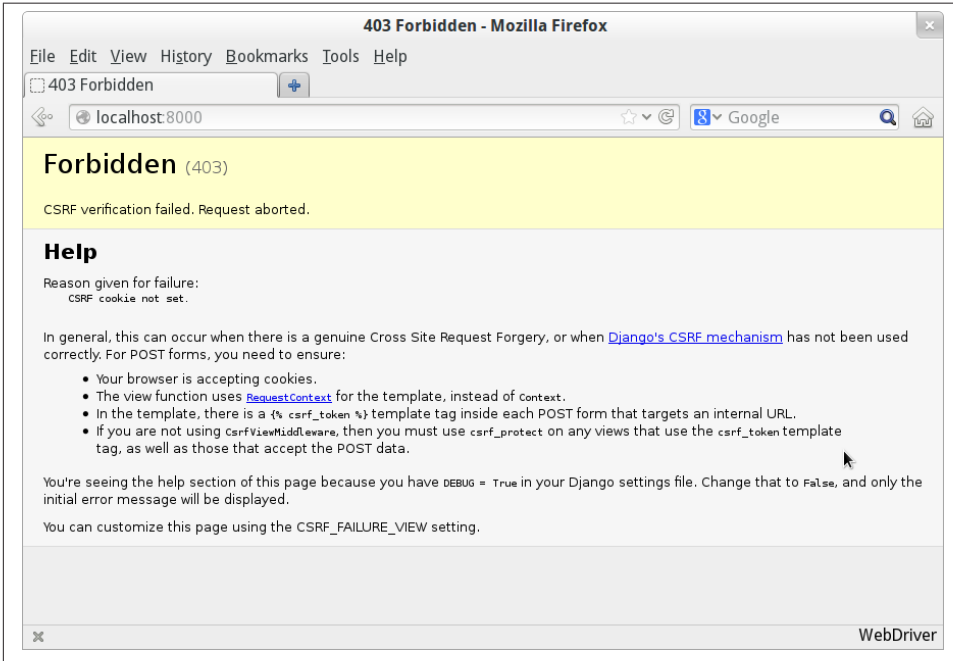


Figure 5-1. Django DEBUG page showing CSRF error

- A brief plug for Ross Anderson’s “Security Engineering” — buy it now!

```

<form method="POST">
    <input id="id_new_item" name="item_text" placeholder="Enter a to-do item" />
    {% csrf_token %}
</form>

```

Re-running the functional test will now give us an expected failure:

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

We can remove the `time.sleep` now though.

```

# "1: Buy peacock feathers" as an item in a to-do list table
inputbox.send_keys(Keys.ENTER)

```



```
table = self.browser.find_element_by_id('id_list_table')
```

Processing a POST request on the server

Add a new method to HomePageTest:

lists/tests.py.

```
def test_home_page_returns_correct_html(self):
    request = HttpRequest()
    response = home_page(request)
    expected_html = render_to_string('home.html')
    self.assertEqual(response.content, expected_html)
```

```
def test_home_page_can_save_a_POST_request(self):
    request = HttpRequest()
    request.method = 'POST'
    request.POST['item_text'] = 'A new list item'

    response = home_page(request)

    self.assertIn('A new list item', response.content)
```

```
$ python manage.py test lists
[...]
AssertionError: 'A new list item' not found in '<html> [...]
```

In typical TDD style, we start with a deliberately silly return value:

lists/views.py.

```
from django.http import HttpResponse
from django.shortcuts import render

def home_page(request):
    if request.method == 'POST':
        return HttpResponse(request.POST['item_text'])
    return render(request, 'home.html')
```

Template context

The syntax `{{ ... }}` lets us display a variable as a string.

lists/templates/home.html.

```
<body>
  <h1>Your To-Do list</h1>
  <form method="POST" >
    <input id="id_new_item" name="item_text" placeholder="Enter a to-do item" />
    {% csrf_token %}
  </form>

  <table id="id_list_table">
    <tr><td>{{ new_item_text }}</td></tr>
  </table>
```

```
</body>
```

lists/tests.py.

```
self.assertIn('A new list item', response.content)
expected_html = render_to_string(
    'home.html',
    {'new_item_text': 'A new list item'})
)
self.assertEqual(response.content, expected_html)
self.assertEqual(response.content, expected_html)
AssertionError: 'A new list item' != u'<html>\n  <head>\n [...]
```

We are allowed to re-write our view, and tell it to pass the POST parameter to the template:

lists/views.py.

```
def home_page(request):
    return render(request, 'home.html', {
        'new_item_text': request.POST['item_text'],
    })
```

Running the unit tests again:

```
ERROR: test_home_page_returns_correct_html (lists.tests.HomePageTest)
[...]
'new_item_text': request.POST['item_text'],
KeyError: 'item_text'
```

An *unexpected failure*... in a different test!

lists/views.py.

```
def home_page(request):
    return render(request, 'home.html', {
        'new_item_text': request.POST.get('item_text', ''),
    })
```

The unit tests should now pass. Let's see what the functional tests say:

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

Hm, not a wonderfully helpful error. Let's use another of our FT debugging techniques: improving the error message.

```
self.assertIn('1: Buy peacock feathers', [row.text for row in rows])
```

functional_tests.py.

Gives

```
AssertionError: '1: Buy peacock feathers' not found in [u'Buy peacock feathers']
```

lists/templates/home.html.

```
<tr><td>1: {{ new_item_text }}</td></tr>
```



ask me about Red/Green/Refactor and Triangulation

Now we get to the `self.fail('Finish the test!')`. We extend our FT.

```
functional_tests.py
# 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)
inputbox = self.browser.find_element_by_id('id_new_item')
inputbox.send_keys('Use peacock feathers to make a fly')
inputbox.send_keys(Keys.ENTER)

# The page updates again, and now shows both items on her list
table = self.browser.find_element_by_id('id_list_table')
rows = table.find_elements_by_tag_name('tr')
self.assertIn('1: Buy peacock feathers', [row.text for row in rows])
self.assertIn(
    '2: Use peacock feathers to make a fly' ,
    [row.text for row in rows]
)

# Edith wonders whether the site will remember her list. Then she sees
# that the site has generate a unique URL for her -- there is some
# explanatory text to that effect.
self.fail('Finish the test!')
```

Now the error is:

```
AssertionError: '1: Buy peacock feathers' not found in [u'1: Use peacock feathers to make a fly']
```

3 strikes and refactor

- code smell

Commit before a refactor:

```
$ git diff
# should show changes to functional_tests.py, home.html,
# tests.py and views.py
$ git commit -a
```

```
functional_tests.py
def check_for_row_in_list_table(self, row_text):
    table = self.browser.find_element_by_id('id_list_table')
    rows = table.find_elements_by_tag_name('tr')
    self.assertIn(row_text, [row.text for row in rows])
```

I like to put helper methods near the top of the class, between the `tearDown` and the first test.

```
functional_tests.py
# 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)
self.check_for_row_in_list_table('1: Buy peacock feathers')
```

```

# 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)
inputbox = self.browser.find_element_by_id('id_new_item')
inputbox.send_keys('Use peacock feathers to make a fly')
inputbox.send_keys(Keys.ENTER)

# The page updates again, and now shows both items on her list
self.check_for_row_in_list_table('1: Buy peacock feathers')
self.check_for_row_in_list_table('2: Use peacock feathers to make a fly')

```

Now we can commit the FT refactor as its own small, atomic change:

```

$ git diff # check the changes to the functional test
$ git commit -a

```

The Django ORM & our first model

Let's create a new class in *lists/tests.py*

lists/tests.py.

```

[...]
from lists.models import Item
from lists.views import home_page
[...]

class ItemModelTest(TestCase):

    def test_saving_and_retrieving_items(self):
        first_item = Item()
        first_item.text = 'The first (ever) list item'
        first_item.save()

        second_item = Item()
        second_item.text = 'Item the second'
        second_item.save()

        saved_items = Item.objects.all()
        self.assertEqual(saved_items.count(), 2)

        first_saved_item = saved_items[0]
        second_saved_item = saved_items[1]
        self.assertEqual(first_saved_item.text, 'The first (ever) list item')
        self.assertEqual(second_saved_item.text, 'Item the second')

```

Let's try running the unit test. Here comes another unit test/code cycle

```

from lists.models import Item
ImportError: cannot import name Item

```

OK then, let's give it something to import from *lists/models.py*.

lists/models.py.

```
from django.db import models
```

```
class Item(object):  
    pass
```

That gets our test as far as:

```
    first_item.save()  
AttributeError: 'Item' object has no attribute 'save'
```

lists/models.py.

```
from django.db import models
```

```
class Item(models.Model):  
    pass
```

Now the test actually gets surprisingly far:

```
    self.assertEqual(first_saved_item.text, 'The first (ever) list item')  
AttributeError: 'Item' object has no attribute 'text'
```

We set up a text field:

lists/models.py.

```
class Item(models.Model):  
    text = models.TextField()
```



ask me about: other field types

```
$ git status  
$ git diff # see changes to tests.py and models.py  
$ git commit -am"Created model for list Items"
```

Saving the POST to the database

We can add 3 new lines (❶) to the existing test called `test_home_page_can_save_a_POST_request`

lists/tests.py.

```
def test_home_page_can_save_a_POST_request(self):  
    request = HttpRequest()  
    request.method = 'POST'  
    request.POST['item_text'] = 'A new list item'  
  
    response = home_page(request)  
  
    self.assertEqual(Item.objects.all().count(), 1) #❶  
    new_item = Item.objects.all()[0]  
    self.assertEqual(new_item.text, 'A new list item')  
  
    self.assertIn('A new list item', response.content)
```

```

expected_html = render_to_string(
    'home.html',
    {'new_item_text': 'A new list item'}
)
self.assertEqual(response.content, expected_html)

```

- code smell
- to-do list

```

self.assertEqual(Item.objects.all().count(), 1)
AssertionError: 0 != 1

```

Let's adjust our view:

```

from django.shortcuts import render
from lists.models import Item

def home_page(request):
    item = Item()
    item.text = request.POST.get('item_text', '')
    item.save()

    return render(request, 'home.html', {
        'new_item_text': request.POST.get('item_text', ''),
    })

```

lists/views.py.

I've coded a very naive solution and you can probably spot a very obvious problem



ask me about ignoring problems like this in real life?

```

return render(request, 'home.html', {
    'new_item_text': item.text
})

```

Our own to-do list:

- Don't save blank items for every request
- Code smell: POST test is too long?
- Display multiple items in the table
- Support more than one list!

Let's start with the first one.

```

def test_home_page_only_saves_items_when_necessary(self):
    request = HttpRequest()

```

lists/tests.py.

```
home_page(request)
self.assertEqual(Item.objects.all().count(), 0)
```

That gives us a `1 != 0` failure.

lists/views.py.

```
def home_page(request):
    if request.method == 'POST':
        new_item_text = request.POST['item_text'] # ❶
        Item.objects.create(text=new_item_text) # ❷
    else:
        new_item_text = '' # ❸

    return render(request, 'home.html', {
        'new_item_text': new_item_text, # ❹
    })
```

That gets the test passing.

Redirect after a POST

But, yuck, that whole `new_item_text = ''` dance is making me pretty unhappy.

lists/tests.py.

```
def test_home_page_can_save_a_POST_request(self):
    request = HttpRequest()
    request.method = 'POST'
    request.POST['item_text'] = 'A new list item'

    response = home_page(request)

    self.assertEqual(Item.objects.all().count(), 1)
    new_item = Item.objects.all()[0]
    self.assertEqual(new_item.text, 'A new list item')

    self.assertEqual(response.status_code, 302)
    self.assertEqual(response['location'], '/')
```

That gives us the error `200 != 302`. We can now tidy up our view substantially:

lists/views.py.

```
from django.shortcuts import redirect, render
from lists.models import Item

def home_page(request):
    if request.method == 'POST':
        Item.objects.create(text=request.POST['item_text'])
        return redirect('/')

    return render(request, 'home.html')
```

And the tests should now pass.

Rendering items in the template

Much better! Back to our to-do list:

- Don't save blank items for every request
- Code smell: POST test is too long?
- Display multiple items in the table
- Support more than one list!

```
def test_home_page_displays_all_list_items(self):  
    Item.objects.create(text='itemey 1')  
    Item.objects.create(text='itemey 2')  
  
    request = HttpRequest()  
    response = home_page(request)  
  
    self.assertIn('itemey 1', response.content)  
    self.assertIn('itemey 2', response.content)
```

lists/tests.py.

That fails as expected:

```
AssertionError: 'itemey 1' not found in '<html>\n  <head>\n [...]
```

lists/templates/home.html.

```
<table id="id_list_table">  
  {% for item in items %}  
    <tr><td>1: {{ item.text }}</td></tr>  
  {% endfor %}  
</table>
```

We need to actually pass the items to it from our home page view:

```
def home_page(request):  
    if request.method == 'POST':  
        Item.objects.create(text=request.POST['item_text'])  
        return redirect('/')  
  
    items = Item.objects.all()  
    return render(request, 'home.html', {'items': items})
```

lists/views.py.

That does get the unit tests to pass... Moment of truth, will the functional test pass?

```
[...] (lots of traceback!)  
AssertionError: 'To-Do' not found in u'ImproperlyConfigured at /'
```

Oops, apparently not. Let's use another functional test debugging technique, and it's one of the most straightforward: manually visiting the site! Open up <http://localhost:8000> in your web browser, and you'll see a Django debug page saying:

```
Please fill out the database NAME in the settings module before using the  
database.
```


Creating our production database with syncdb

superlists/settings.py.

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.sqlite3',  
        'NAME': 'database.sqlite',  
    }  
    [...]
```

If we try reloading the page on localhost at this point, it will tell us that there is a `DatabaseError`, no such table: `lists_item`.

```
$ python manage.py syncdb  
Creating tables ...  
Creating table auth_permission  
Creating table auth_group_permissions  
Creating table auth_group  
Creating table auth_user_user_permissions  
Creating table auth_user_groups  
Creating table auth_user  
Creating table django_content_type  
Creating table django_session  
Creating table django_site  
Creating table lists_item
```

You just installed Django's auth system, which means you don't have any superusers defined.

Would you like to create one now? (yes/no): **no**

Installing custom SQL ...

Installing indexes ...

Installed 0 object(s) from 0 fixture(s)

We can refresh the page on *localhost*, see that our error is gone, and try running the functional tests again.

```
AssertionError: '2: Use peacock feathers to make a fly' not found in  
[u'1: Buy peacock feathers', u'1: Use peacock feathers to make a fly']
```

Oooh, so close! We just need to get our list numbering right. Another awesome Django template tag will help here: `forloop.counter`:

```
{% for item in items%}  
    <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>  
{% endfor %}
```

lists/templates/home.html.

If you try it again, you should now see the FT get to the end:

```
self.fail('Finish the test!')  
AssertionError: Finish the test!
```

But, as it's running, you may notice something is amiss

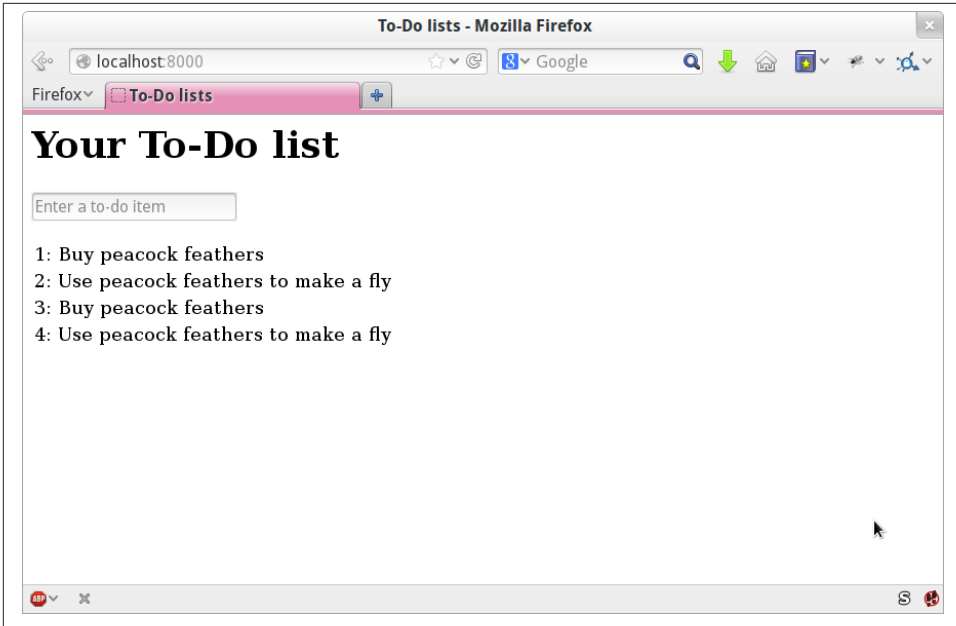


Figure 5-2. There are list items left over from the last run of the test

```
1: Buy peacock feathers
2: Use peacock feathers to make a fly
3: Buy peacock feathers
4: Use peacock feathers to make a fly
5: Buy peacock feathers
6: Use peacock feathers to make a fly

$ rm database.sqlite
$ python manage.py syncdb # "no" to superuser again

$ git add lists
$ git commit -m"Redirect after POST, and show all items in template"
$ git add superlists/settings.py
$ echo "database.sqlite" >> .gitignore
$ git add .gitignore
$ git commit -m"Name database in settings.py, add it to .gitignore"
```

Where are we?

- We've got a form set up to add new items to the list using POST.
- We've set up a simple model in the database to save list items.
- We've used at least 3 different FT debugging techniques.

But we've got a couple of items on our own to-do list, namely getting the FT to clean up after itself, and perhaps more critically, adding support for more than one list.

I mean, we *could* ship the site as it is, but people might find it strange that the entire human population has to share a single to-do list. I suppose it might get people to stop and think about how connected we all are to one another, how we all share a common destiny here on spaceship Earth, and how we must all work together to solve the global problems that we face.

But, in practical terms, the site wouldn't be very useful...

Ah well.

Useful TDD concepts

Regression

When new code breaks some aspect of the application which used to work.

Unexpected failure

When a test fails in a way we weren't expecting. This either means that we've made a mistake in our tests, or that the tests have helped us find a regression, and we need to fix something in our code.

Red / Green / Refactor

Another way of describing the TDD process. Write a test and see it fail (Red), write some code to get it to pass (Green), then Refactor to improve the implementation.

Triangulation

The act of writing extra test code in order to make sure that our implementation is correct.

3 strikes and refactor

A rule of thumb for when to remove duplication from code.

The scratchpad to-do list

A place to write down things that occur to us as we're coding, so that we can finish up what we're doing and come back to them later.

Getting to the minimum viable site

- i. In which we use incremental, step-by-step refactoring to get to a better app. Testing Goat, not Refactoring Cat!

Ensuring test isolation in functional tests



ask me about “rolling your own” database cleanup

```
$ mkdir functional_tests
$ touch functional_tests/__init__.py
$ touch functional_tests/models.py
```

Then move the FT file:

```
$ git mv -f functional_tests.py functional_tests/tests.py
$ git status # should show the "rename" and two untracked files
```

At this point your directory tree should look like this:

```
├─ database.sqlite
├─ functional_tests
│  ├─ __init__.py
│  ├─ models.py
│  └─ tests.py
├─ lists
│  ├─ __init__.py
│  ├─ models.py
│  └─ templates
│     └─ home.html
└─ tests.py
```

```

├── views.py
├── manage.py
├── superlists
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py

```



ask me about — where to put FTs?

functional_tests/tests.py.

```

from django.test import LiveServerTestCase #1
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

class NewVisitorTest(LiveServerTestCase): #2

    def setUp(self):
        self.browser = webdriver.Firefox()
        self.browser.implicitly_wait(3)

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

    def check_for_row_in_list_table(self, row_text):
        [...]

    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(self.live_server_url) #3

        [...]
# also remove the if __name__ == '__main__' from the end #4

```

Finally, we add `functional_tests` as a new app in `superlists/settings.py`:

superlists/settings.py.

```

INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # Uncomment the next line to enable the admin:
    # 'django.contrib.admin',
    # Uncomment the next line to enable admin documentation:
    # 'django.contrib.admindocs',

```

```

    'lists',
    'functional_tests',
)

```

Now we are able to run our Functional tests using the Django test runner, by telling it to run just the tests for our new `functional_tests` app:

```

$ python manage.py test functional_tests
Creating test database for alias 'default'...
F
=====
FAIL: test_can_start_a_list_and_retrieve_it_later (functional_tests.tests.NewVisitorTest)
-----
Traceback (most recent call last):
  File "/media/SHARED/Dropbox/book/source/chapter_6/superlists/functional_tests/tests.py", line 74:
    self.fail('Finish the test!')
AssertionError: Finish the test!

-----
Ran 1 test in 6.378s

FAILED (failures=1)
Destroying test database for alias 'default'...

```



If, before the FAIL, you see some traceback ending in a `TemplateDoesNotExist: 500`, it's because you're running Django 1.4 instead of 1.5. You should upgrade, because although not much has changed between the two versions, a few subtle things like this have. It's not too late to upgrade at this stage.

Success! We should commit it as an atomic change:

```

$ git status # should show renamed functional_tests.py,
              # modified functional_tests/tests.py & settings.py
              # and 2 new files, __init__.py and models.py
$ git add functional_tests
$ git add superlists/settings.py
$ git diff --staged -M
$ git commit # msg eg "move functional_tests to functional_tests app, use LiveServerTestCase"

```

The `-M` flag on the `git diff` is a useful one. It means “detect moves”, so it will notice that `functional_tests.py` and `functional_tests/tests.py` are the same file, and show you a more sensible diff (try it without!).

Useful commands updated

To run the functional tests

```
python manage.py test functional_tests
```

To run the unit tests

```
python manage.py test lists
```

Currently the FT says this:

```
functional_tests/tests.py.  
# Edith wonders whether the site will remember her list. Then she sees  
# that the site has generate a unique URL for her -- there is some  
# explanatory text to that effect.  
self.fail('Finish the test!')  
  
# She visits that URL - her to-do list is still there.  
  
# Satisfied, she goes back to sleep
```

Let's think about this a bit more.

Small Design When Necessary

- Big Design up-front
- Minimum viable app
- YAGNI

REST

...ish

Each list can have its own URL, like

```
/lists/<list identifier>/
```

To create a brand new list, we'll have a special URL that accepts POST requests:

```
/lists/new
```

To add a new item to an existing list, we'll have a separate URL, to which we can send POST requests.

```
/lists/<list identifier>/add_item
```

In summary, our scratchpad for this chapter looks something like this:

- ~~Get FTs to clean up after themselves~~
- Adjust model so that items are associated with different lists
- Add unique URLs for each list
- Add a URL for creating a new list via POST

- Add URLs for adding a new item to an existing list via POST

Implementing the new design using TDD

Look for the point at which we say `inputbox.send_keys('Buy peacock feathers')`, and amend the next block of code like this:

```
inputbox.send_keys('Buy peacock feathers') functional_tests/tests.py.

# When she hits enter, she is taken to a new URL,
# and now the page lists "1: Buy peacock feathers" as an item in a
# to-do list table
inputbox.send_keys(Keys.ENTER)
edith_list_url = self.browser.current_url
self.assertRegexpMatches(edith_list_url, '/lists/.+')
self.check_for_row_in_list_table('1: Buy peacock feathers')
[...]
```

Let's change the end of the test and imagine a new user coming along. Delete everything from the comments just before the `self.fail` (they say "Edith wonders whether the site will remember her list..."), and replace them with a new ending to our FT:

```
[...] functional_tests/tests.py.
# The page updates again, and now shows both items on her list
self.check_for_row_in_list_table('2: Use peacock feathers to make a fly')
self.check_for_row_in_list_table('1: Buy peacock feathers')

# Now a new user, Francis, comes along to the site.
self.browser.quit()
## We use a new browser session to make sure that no information
## of Edith's is coming through from cookies etc #1
self.browser = webdriver.Firefox()

# Francis visits the home page. There is no sign of Edith's
# list
self.browser.get(self.live_server_url)
page_text = self.browser.find_element_by_tag_name('body').text
self.assertNotIn('Buy peacock feathers', page_text)
self.assertNotIn('make a fly', page_text)

# Francis starts a new list by entering a new item. He
# is less interesting than Edith...
inputbox = self.browser.find_element_by_id('id_new_item')
inputbox.send_keys('Buy milk')
inputbox.send_keys(Keys.ENTER)

# Francis gets his own unique URL
francis_list_url = self.browser.current_url
self.assertRegexpMatches(francis_list_url, '/lists/.+') #2
```

```

self.assertNotEqual(francis_list_url, edith_list_url)

# Again, there is no trace of Edith's list
page_text = self.browser.find_element_by_tag_name('body').text
self.assertNotIn('Buy peacock feathers', page_text)
self.assertIn('Buy milk', page_text)

```

We run our FTs:

```

AssertionError: Regexp didn't match: '/lists/.+' not found in
u'http://localhost:8081/'

$ git commit -a

```

Iterating towards the new design

The URL comes from the redirect after POST. In *lists/tests.py*, find `test_home_page_can_save_a_POST_request`, and change the expected redirect location:

```

self.assertEqual(response.status_code, 302)
self.assertEqual(response['location'], '/lists/the-only-list-in-the-world/')

```

lists/tests.py.

- Want to solve for N? Solve for 1 first.

```

$ python manage.py test lists
[...]
AssertionError: '/' != '/lists/the-only-list-in-the-world/'

```

Now we can go adjust our `home_page` view in *lists/views.py*:

```

def home_page(request):
    if request.method == 'POST':
        Item.objects.create(text=request.POST['item_text'])
        return redirect('/lists/the-only-list-in-the-world/')

```

lists/views.py.

Of course that will now totally break the functional test, because there is no such URL on our site yet. So, let's build a special URL for our one and only list.

Testing views, templates and URLs together with the Django Test Client

```

from django.test import Client, TestCase
[...]

class ListViewTest(TestCase):

    def test_list_view_displays_all_items(self):

```

lists/tests.py.

```

Item.objects.create(text='itemey 1')
Item.objects.create(text='itemey 2')

client = Client()
response = client.get('/lists/the-only-list-in-the-world/')

self.assertIn('itemey 1', response.content)
self.assertIn('itemey 2', response.content)

```

Let's try running the test now:

```

self.assertIn('itemey 1', response.content)
AssertionError: 'itemey 1' not found in '<h1>Not Found</h1><p>The requested URL
/lists/the-only-list-in-the-world/ was not found on this server.</p>'

```

Our singleton list URL doesn't exist yet. We fix that in *superlists/urls.py*

```

urlpatterns = patterns('',
    url(r'^$', 'lists.views.home_page', name='home'),
    url(r'^lists/the-only-list-in-the-world/$', 'lists.views.view_list',
        name='view_list'
    ),
    # url(r'^superlists/', include('superlists.foo.urls')),
    [...])

```

superlists/urls.py.

Running the tests again, we get:

```

ValueError: Could not import lists.views.view_list. View does not exist
in module lists.views.

```

Nicely self-explanatory. Let's create a dummy view function in *lists/views.py*

```

def view_list(request):
    pass

```

lists/views.py.

Now we get

```

ValueError: The view lists.views.view_list didn't return an HttpResponse object.

```

Let's copy the two last lines from the *home_page* view and see if they'll do the trick:

```

def view_list(request):
    items = Item.objects.all()
    return render(request, 'home.html', {'items': items})

```

lists/views.py.

Re-run the tests and they should pass:

```

Ran 7 tests in 0.052s
OK

```

And the FTs should get a little further on:

```

AssertionError: '2: Use peacock feathers to make a fly' not found in [u'1: Buy
peacock feathers']

```

Time for a little tidying up.

```

$ egrep "class|def" lists/tests.py
class HomePageTest(TestCase):
    def test_root_url_resolves_to_home_page_view(self):
    def test_home_page_returns_correct_html(self):
    def test_home_page_displays_all_list_items(self):
    def test_home_page_only_saves_items_when_necessary(self):
    def test_home_page_can_save_a_POST_request(self):
class ListViewTest(TestCase):
    def test_list_view_displays_all_items(self):
class ItemModelTest(TestCase):
    def test_saving_and_retrieving_items(self):

```

Start by deleting the `test_home_page_displays_all_list_items` method. If you run `manage.py test lists` now, it should say it ran 6 tests instead of 7.

A new template:

```

class ListViewTest(TestCase):
    def test_list_view_displays_all_items(self):
        Item.objects.create(text='itemey 1')
        Item.objects.create(text='itemey 2')

        client = Client()
        response = client.get('/lists/the-only-list-in-the-world/')

        self.assertIn('itemey 1', response.content)
        self.assertIn('itemey 2', response.content)
        self.assertTemplateUsed(response, 'list.html')

```

lists/tests.py.

Let's see what it says:

```

AssertionError: Template 'list.html' was not a template used to render the
response. Actual template(s) used: home.html

```

Great! Let's change the view:

```

def view_list(request):
    items = Item.objects.all()
    return render(request, 'list.html', {'items': items})

```

lists/views.py.

But, obviously, that template doesn't exist yet. If we run the unit tests, we get:

```

TemplateDoesNotExist: list.html

```

Let's create a new file at `lists/templates/list.html`.

```

$ touch lists/templates/list.html

```

A blank template, which gives us this error — good to know the tests are there to make sure we fill it in:

```

AssertionError: 'itemey 1' not found in ''

```

The template for an individual list will re-use quite a lot of the stuff we currently have in *home.html*, so we can start by just copying that:

```
$ cp lists/templates/home.html lists/templates/list.html
```

That gets the tests back to passing (green). Now let's do a little more tidying up (refactoring). We said the home page doesn't need to list items, it only needs the new list input field, so we can remove some lines from *lists/templates/home.html*, and maybe slightly tweak the h1 to say "Start a new list":

```
lists/templates/home.html.  
  
<html>  
  <head>  
    <title>To-Do lists</title>  
  </head>  
  <body>  
    <h1>Start a To-Do list</h1>  
    <form method="POST" >  
      <input id="id_new_item" name="item_text" placeholder="Enter a to-do item" />  
      {% csrf_token %}  
    </form>  
  </body>  
</html>
```

```
lists/views.py.  
  
def home_page(request):  
    if request.method == 'POST':  
        Item.objects.create(text=request.POST['item_text'])  
        return redirect('/lists/the-only-list-in-the-world/')  
    return render(request, 'home.html')
```

```
lists/templates/list.html.  
  
<h1>Your To-Do list</h1>
```

Let's run the functional tests:

```
AssertionError: '2: Use peacock feathers to make a fly' not found in [u'1: Buy  
peacock feathers']
```

The `action=` attribute...

```
lists/templates/list.html.  
  
<form method="POST" action="/" >
```

And try running the FT again:

```
self.assertNotEqual(francis_list_url, edith_list_url)  
AssertionError: u'http://localhost:8081/lists/the-only-list-in-the-world/' ==  
u'http://localhost:8081/lists/the-only-list-in-the-world/'
```

Hooray! We're back to where we were earlier, which means our refactoring is complete — we now have a unique URL for our one list.

```
$ git status # should show 4 changed files and 1 new file, list.html  
$ git add lists/templates/list.html  
$ git diff # should show we've simplified home.html,  
           # moved one test to a new class in lists/tests.py added a new view
```

```
# in views.py, and simplified home_page and made one addition to
# urls.py
$ git commit -a # add a message summarising the above, maybe something like
# "new URL, view and template to display lists"
```

Adding another URL

Let's take a look at our to-do list:

- Get FTs to clean up after themselves
- Adjust model so that items are associated with different lists
- Add unique URLs for each list
- Add a URL for creating a new list via POST
- Add URLs for adding a new item to an existing list via POST

Open up *lists/tests.py*, and *move* the *test_home_page_can_save_a_POST_request* method into a new class, then change its name:

```
[...] lists/tests.py.
self.assertEqual(Item.objects.all().count(), 0)
```

```
class NewListTest(TestCase):

    def test_saving_a_POST_request(self):
        request = HttpRequest()
        request.method = 'POST'
        [...]
```

Now let's use the Django test client:

```
class NewListTest(TestCase): lists/tests.py.

    def test_saving_a_POST_request(self):
        client = Client()
        response = client.post(
            '/lists/new',
            data={'item_text': 'A new list item'}
        )

        self.assertEqual(Item.objects.all().count(), 1)
        new_item = Item.objects.all()[0]
        self.assertEqual(new_item.text, 'A new list item')

        self.assertEqual(response.status_code, 302)
        self.assertEqual(response['location'], '/lists/the-only-list-in-the-world/')
```

Let's try running that:

```
self.assertEqual(Item.objects.all().count(), 1)
AssertionError: 0 != 1
```

Hmmm, a little baffling. I think I know why. Let's move the `response.status_code` check a little higher up:

```
def test_saving_a_POST_request(self):
    client = Client()
    response = client.post(
        '/lists/new',
        data={'item_text': 'A new list item'}
    )
    self.assertEqual(response.status_code, 302)

    self.assertEqual(Item.objects.all().count(), 1)
    new_item = Item.objects.all()[0]
    self.assertEqual(new_item.text, 'A new list item')

    self.assertEqual(response['location'], '/lists/the-only-list-in-the-world/')
```

Aha! Sure enough:

```
self.assertEqual(response.status_code, 302)
AssertionError: 404 != 302
```

We haven't built a URL for `/lists/new`, so the `client.post` is just getting a 404 response. Let's build it now.

```
urlpatterns = patterns('',
    url(r'^$', 'lists.views.home_page', name='home'),
    url(r'^lists/the-only-list-in-the-world/$', 'lists.views.view_list',
        name='view_list'
    ),
    url(r'^lists/new$', 'lists.views.new_list', name='new_list'),
```

Next we get a `ViewDoesNotExist`, so let's fix that, in `lists/views.py`:

```
def new_list(request):
    pass
```

Then we get "The view `lists.views.new_list` didn't return an `HttpResponse` object." (this is getting rather familiar!). Let's borrow a line from `home_page`

```
def new_list(request):
    return redirect('/lists/the-only-list-in-the-world/')
    self.assertEqual(Item.objects.all().count(), 1)
AssertionError: 0 != 1
```

And another line from `home_page`:

lists/views.py.

```
def new_list(request):  
    Item.objects.create(text=request.POST['item_text'])  
    return redirect('/lists/the-only-list-in-the-world/')
```

Oops, an unexpected fail:

```
self.assertEqual(response['location'], '/lists/the-only-list-in-the-world/')  
AssertionError: 'http://testserver/lists/the-only-list-in-the-world/' !=  
'/lists/the-only-list-in-the-world/'
```

Let's use another of Django's test helper functions instead of our two-step check for the redirect:

lists/tests.py.

```
def test_saving_a_POST_request(self):  
    client = Client()  
    response = client.post(  
        '/lists/new',  
        data={'item_text': 'A new list item'}  
    )  
  
    self.assertEqual(Item.objects.all().count(), 1)  
    new_item = Item.objects.all()[0]  
    self.assertEqual(new_item.text, 'A new list item')  
  
    self.assertRedirects(response, '/lists/the-only-list-in-the-world/')
```

That now passes. We're looking good. Can we remove the old `if request.method == 'POST'` code from `home_page`?

lists/views.py.

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

Doesn't that feel good? The view functions are looking much simpler. We re-run the tests to make sure... Yes, 6 tests OK.

Finally, let's wire up our two forms to use this new URL. In *both* `home.html` and `lists.html`:

lists/templates/home.html, lists/templates/list.html.

```
<form method="POST" action="/lists/new" >
```

And we re-run our FTs to make sure everything still works...

```
AssertionError: u'http://localhost:8081/lists/the-only-list-in-the-world/' ==  
u'http://localhost:8081/lists/the-only-list-in-the-world/'  
  
$ git status # 5 changed files  
$ git diff # URLs for forms x2, moved code in views + tests, new URL  
$ git commit -a
```

Adjusting our models

A diff output instead of a plain code listing:


```

-from lists.models import Item
+from lists.models import Item, List
from lists.views import home_page

@@ -62,14 +62,21 @@ class ListViewTest(TestCase):
-class ItemModelTest(TestCase):
+class ListAndItemModelsTest(TestCase):

    def test_saving_and_retrieving_items(self):
+    list = List()
+    list.save()

        first_item = Item()
        first_item.text = 'The first (ever) list item'
+    first_item.list = list
        first_item.save()

        second_item = Item()
        second_item.text = 'Item the second'
+    second_item.list = list
        second_item.save()

+    saved_lists = List.objects.all()
+    self.assertEqual(saved_lists.count(), 1)
+    self.assertEqual(saved_lists[0], list)
        saved_items = Item.objects.all()
        self.assertEqual(saved_items.count(), 2)

        first_saved_item = saved_items[0]
        second_saved_item = saved_items[1]
        self.assertEqual(first_saved_item.text, 'The first (ever) list item')
+    self.assertEqual(first_saved_item.list, list)
        self.assertEqual(second_saved_item.text, 'Item the second')
+    self.assertEqual(second_saved_item.list, list)

```

Time for another unit-test/code cycle. I'm just going to show the test errors for the first couple, and let you figure out for yourself what the code should be:

```

ImportError: cannot import name List

AttributeError: 'List' object has no attribute 'save'

    self.assertEqual(first_saved_item.list, list)
AttributeError: 'Item' object has no attribute 'list'

```

How do we give our Item a list attribute?

```

class Item(models.Model):
    text = models.TextField()
    list = models.TextField()

```

That give us:

```
AssertionError: u'List object' != <List: List object>
```

lists/models.py.

```
class List(models.Model):
    pass

class Item(models.Model):
    text = models.TextField()
    list = models.ForeignKey(List)
```

Now what happens?

```
$ python manage.py test functional_tests
Creating test database for alias 'default'...
....EE
=====
ERROR: test_list_view_displays_all_items (lists.tests.ListViewTest)
-----
Traceback (most recent call last):
  File "/media/SHARED/Dropbox/book/source/chapter_6/superlists/lists/tests.py",
line 50, in test_list_view_displays_all_items
    Item.objects.create(text='itemey 1')
    [...]
    return Database.Cursor.execute(self, query, params)
IntegrityError: lists_item.list_id may not be NULL

=====
ERROR: test_saving_a_POST_request (lists.tests.NewListTest)
-----
Traceback (most recent call last):
  File "/media/SHARED/Dropbox/book/source/chapter_6/superlists/lists/tests.py",
line 36, in test_saving_a_POST_request
    data={'item_text': 'A new list item'}
    [...]
    return Database.Cursor.execute(self, query, params)
IntegrityError: lists_item.list_id may not be NULL

-----
Ran 6 tests in 0.017s
```

Oh gawd! Still, this is exactly why we have tests.

lists/tests.py.

```
class ListViewTest(TestCase):

    def test_list_view_displays_all_items(self):
        list = List.objects.create()
        Item.objects.create(text='itemey 1', list=list)
        Item.objects.create(text='itemey 2', list=list)
```

That gets us down to one failing test. Decoding its traceback, it fails in the view:

```
File "/media/SHARED/Dropbox/book/source/chapter_6/superlists/lists/views.py",
line 9, in new_list
    Item.objects.create(text=request.POST['item_text'])
```

So we make a similar change in the view:

```
from lists.models import Item, List
[...]
def new_list(request):
    list = List.objects.create()
    Item.objects.create(text=request.POST['item_text'], list=list)
    return redirect('/lists/the-only-list-in-the-world/')
```

lists/views.py.

Are you cringing internally at this point?

Anyway, just to reassure ourselves that things have worked, we can re-run the FT.

```
$ git status # 3 changed files
$ git diff
$ git commit -a
```

The final stage: each list should have its own URL

```
class ListViewTest(TestCase):

    def test_list_view_displays_items_for_that_list(self):
        list = List.objects.create()
        Item.objects.create(text='itemey 1', list=list)
        Item.objects.create(text='itemey 2', list=list)

        other_list = List.objects.create()
        Item.objects.create(text='other list item 1', list=other_list)
        Item.objects.create(text='other list item 2', list=other_list)

        client = Client()
        response = client.get('/lists/%d/' % (list.id,))

        self.assertIn('itemey 1', response.content)
        self.assertIn('itemey 2', response.content)
        self.assertNotIn('other list item 1', response.content)
        self.assertNotIn('other list item 2', response.content)
        self.assertTemplateUsed(response, 'list.html')
```

lists/tests.py.

Running the unit tests gives us:

```
AssertionError: 'itemey 1' not found in '404 Page not found. Try another
URL.\n'
```

It's time to learn how we can pass parameters from URLs to views:

```
urlpatterns = patterns('',
    url(r'^$', 'lists.views.home_page', name='home'),
    url(r'^lists/(.+)/$', 'lists.views.view_list', name='view_list'),
    url(r'^lists/new$', 'lists.views.new_list', name='new_list'),
```

superlists/urls.py.

But our view doesn't expect an argument yet!

```
ERROR: test_list_view_displays_all_items (lists.tests.ListViewTest)
[...]
TypeError: view_list() takes exactly 1 argument (2 given)
```

```
ERROR: test_saving_a_POST_request (lists.tests.NewListTest)
[...]
TypeError: view_list() takes exactly 1 argument (2 given)
```

We can fix that easily with a dummy parameter in *views.py*

```
def view_list(request, list_id):
```

lists/views.py.

Now we're down to our expected failure:

```
self.assertNotIn('other list item 1', response.content)
AssertionError: 'other list item 1' unexpectedly found in [...]
```

Let's make our view discriminate over which items it sends to the template:

```
def view_list(request, list_id):
    list = List.objects.get(id=list_id)
    items = Item.objects.filter(list=list)
    return render(request, 'list.html', {'items': items})
```

lists/views.py.

Now we get an error in another test:

```
ERROR: test_saving_a_POST_request (lists.tests.NewListTest)
self.assertRedirects(response, '/lists/the-only-list-in-the-world/')
[...]
return int(value)
ValueError: invalid literal for int() with base 10: 'the-only-list-in-the-world'
```

Let's take a look at this test then, since it's whining. This brings to mind the fact that we actually need to treat the creation of *new* lists differently from the addition of new items to *existing* lists.

```
self.assertEqual(List.objects.all().count(), 1)
new_list = List.objects.all()[0]
self.assertEqual(Item.objects.all().count(), 1)
new_item = Item.objects.all()[0]
self.assertEqual(new_item.text, 'A new list item')
self.assertEqual(new_item.list, new_list)

self.assertRedirects(response, '/lists/%d/' % (new_list.id,))
```

lists/tests.py.

That still gives us the *invalid literal* error.

```
return redirect('/lists/%d/' % (list.id,))
```

lists/views.py.

That gets us back to passing unit tests. What about the functional tests? We must be almost there?

```
AssertionError: '2: Use peacock feathers to make a fly' not found in
[u'1: Use peacock feathers to make a fly']
```

This is exactly what we have functional tests for!

We need a URL and view to handle adding a new item to an existing list.

```
class NewItemTest(TestCase):
    def test_saving_a_POST_request_to_an_existing_list(self):
        list = List.objects.create()
        other_list = List.objects.create()
        client = Client()
        response = client.post(
            '/lists/%d/new_item' % (list.id,),
            data={'item_text': 'A new item for an existing list'}
        )

        self.assertEqual(Item.objects.all().count(), 1)
        new_item = Item.objects.all()[0]
        self.assertEqual(new_item.text, 'A new item for an existing list')
        self.assertEqual(new_item.list, list)

        self.assertRedirects(response, '/lists/%d/' % (list.id,))
```

We get

```
AssertionError: 0 != 1
```

This is because the view is actually giving a 404 (again, you can check by moving the `assertRedirects` higher in the test if you like)

Add a new URL in `urls.py`:

```
urlpatterns = patterns('',
    url(r'^$', 'lists.views.home_page', name='home'),
    url(r'^lists/(.+)/$', 'lists.views.view_list', name='view_list'),
    url(r'^lists/(.+)/new_item$', 'lists.views.add_item', name='add_item'),
    url(r'^lists/new$', 'lists.views.new_list', name='new_list'),
```

Hm, three very similar-looking URLs there. Let's make a note on our to-do list, they look like good candidates for a refactoring.

- Get FTs to clean up after themselves
- Adjust model so that items are associated with different lists
- Add unique URLs for each list
- Add a URL for creating a new list via POST
- Add URLs for adding a new item to an existing list via POST
- Refactor away some duplication in `urls.py`

Back to the tests, we now get:

ViewDoesNotExist: Could not import lists.views.add_item. View does not exist in module lists.views.

Let's try:

```
def add_item(request):  
    pass
```

lists/views.py.

Aha:

TypeError: add_item() takes exactly 1 argument (2 given)

```
def add_item(request, list_id):  
    pass
```

lists/views.py.

And then:

ValueError: The view lists.views.add_item didn't return an HttpResponse object.

let's copy the redirect from new_list and the List.objects.get from view_list:

```
def add_item(request, list_id):  
    list = List.objects.get(id=list_id)  
    return redirect('/lists/%d/' % (list.id,))  
    self.assertEqual(Item.objects.all().count(), 1)  
AssertionError: 0 != 1
```

lists/views.py.

And finally let's make it save our new list item:

```
def add_item(request, list_id):  
    list = List.objects.get(id=list_id)  
    Item.objects.create(text=request.POST['item_text'], list=list)  
    return redirect('/lists/%d/' % (list.id,))
```

lists/views.py.

Now we just need to use this URL in our *list.html* template. Open it up and adjust the form tag...

```
<form method="POST" action="but what should we put here?" >
```

lists/templates/list.html.

```
self.assertNotIn('other list item 2', response.content)  
self.assertTemplateUsed(response, 'list.html')  
self.assertEqual(response.context['list'], list)
```

lists/tests.py.

That gives us `KeyError: 'list'`

```
def view_list(request, list_id):  
    list = List.objects.get(id=list_id)  
    return render(request, 'list.html', {'list': list})
```

lists/views.py.

That, of course, will break because the template is expecting `items`, but we can fix it in *list.html*

lists/templates/list.html.

```

<form method="POST" action="/lists/{{ list.id }}/new_item" >

[...]

{% for item in list.item_set.all %}
  <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
{% endfor %}

```

So that gets the unit tests to pass. How about the FT?

```

$ python manage.py test functional_tests
Creating test database for alias 'default'...
.
-----
Ran 1 test in 5.824s

OK

```

YES! And a quick check on our to-do list:

- ~~Adjust model so that items are associated with different lists~~
- Add unique URLs for each list
- Add a URL for creating a new list via POST
- Add URLs for adding a new item to an existing list via POST
- Refactor away some duplication in *urls.py*

Irritatingly, the Testing Goat is a stickler for tying up loose ends too, so we've got to do this one final thing.

Before we start, we'll do a commit - always make sure you've got a commit of a working state before embarking on a refactor

```

$ git diff
$ git commit -am"new URL + view for adding to existing lists. FT passes!"

```

A final refactor using URL includes

```

$ cp superlists/urls.py lists/

```

superlists/urls.py.

```

urlpatterns = patterns('',
    url(r'^$', 'lists.views.home_page', name='home'),
    url(r'^lists/', include('lists.urls')),

    # Uncomment the admin/doc line below to enable admin documentation:
    # url(r'^admin/doc/', include('django.contrib.admindocs.urls')),

    # Uncomment the next line to enable the admin:
    # url(r'^admin/', include(admin.site.urls)),
)

```

```
from django.conf.urls import patterns, url

urlpatterns = patterns('',
    url(r'^(.+)/$', 'lists.views.view_list', name='view_list'),
    url(r'^(.+)/new_item$', 'lists.views.add_item', name='add_item'),
    url(r'^new$', 'lists.views.new_list', name='new_list'),
)

$ git status
$ git add lists/urls.py
$ git add superlists/urls.py
$ git diff --staged
$ git commit
```

Phew. A marathon chapter. But we covered a number of important topics, starting with test isolation, and then some thinking about design. We saw how to adapt an existing site step-by-step, going from working state to working state, in order to iterate towards our new REST-ish structure. We covered some rules of thumb like “YAGNI” and “3 strikes then refactor”

Useful TDD concepts

Test isolation

Different tests shouldn't affect one another. This means we need to reset any permanent state at the end of each test. Django's test runner helps us do this by creating a test database, which it wipes clean in between each test.

The Testing Goat vs Refactoring Cat

Our natural urge is often to dive in and fix everything at once... but if we're not careful, we'll end up like Refactoring Cat, in a situation with loads of changes to our code and nothing working. The Testing Goat encourages us to take on step at a time, and go from working state to working state.

Outline to date & future chapters plan

Thanks for reading this far! I'd really like your input on this too: What do you think of the book so far, and what do you think about the topics I'm proposing to cover in the list below? Email me at obeythetestinggoat@gmail.com!

Chapter 0 / Preface

BOOK 1: Building a minimum viable app with TDD

Already done:

- Chapter 1 - Getting Django set up using a Functional Test
- Chapter 2 - Extending our FT using the unittest module
- Chapter 3 - Testing a simple home page with unit tests
- Chapter 4 - What are we doing with all these tests?
- Chapter 5 - Saving form submissions to the database
- Chapter 6 - Getting to the minimum viable site
- Chapter 7 - Prettification
- Chapter 8 - Deploy!

BOOK 2: Growing the site

Chapter 9: User Authentication + the admin site

- users want to be able to view *their* todos
- related field on list for owner

- FT Sign up, login/logout
- Explain the admin site
- Fixtures?
- “claim” an existing list (?)
- URLs would need to be less guessable

Chapter 10: A more complex model, forms and validation

- mark completed
- notes field
- validation: check that items aren't duplicates

Chapter 11: javascript

- show / hide notes field
- choose JS testing framework (QUnit, YUI / other?)

Chapter 11: Ajax

- use markdown for notes
- dynamic previews, like on SO

Chapter 12: sharing lists

- email notifications
- django notifications (?)

Chapter 13: oauth

- passwords suck!
- mocking external web service to check if broken

More/Other possible contents

Django stuff:

- pagination
- switch database to eg postgres
- South
- FT for 404 and 500 pages

Testing topics to work in

- how to read a traceback (prob. insert back into an earlier chapter)
- snapshot_on_error
- what to test in templates
- fixtures (factory boy?)
- mocking (python mock)
- selenium page pattern
- JS: mocking external web service to simulate errors
- coverage
- difference between unittest.TestCase, django.test.TestCase, LiveServerTestCase
- Splinter
- addCleanup

External systems integration

- gravatar
- Mozilla persona?

Deployment stuff

- South + testing data migrations
- FT for 404 and 500 pages?
- email integration

BOOK 3: Trendy stuff

Chapter 14: CI

Chapter 15 & 16: More Javascript

- MVC tool (backbone / angular)

- single page website (?) or bottomless web page?
- switching to a full REST API
- HTML5, eg LocalStorage
- Encryption - client-side decrypt lists, for privacy?

Chapter 17: Async

- websockets
- tornado/gevent (or sthing based on Python 3 async??)

Chapter 18: NoSQL

- obligatory discussion of NoSQL and MongoDB
- describe installation, particularities of testing

Chapter 19: Caching

- unit testing memcached
- Functionally testing performance
- Apache ab testing

5/6 chapters?

Appendices

Other possible appendix(?) topics

- Deployment. Discuss a few options — pythonanywhere, heroku, ec2
- BDD (+2 from reddit)
- Mobile (use selenium, link to using bootstrap?)
- Payments... Some kind of shopping cart?
- unit testing fabric scripts

Existing appendix I: PythonAnywhere

- Running Firefox Selenium sessions with pyVirtualDisplay
- Setting up Django as a PythonAnywhere web app
- Cleaning up /tmp
- Screenshots