Test-Driven Development (TDD) simplifies the trickiest of software tasks with its unique ability to peel back problems into layers. The testing tools available in Python and Django make test writing a joy, and the full coverage test suite that results from TDD is a boon to any project.

This guide to developing with Django takes a test-first approach: write a test, then write enough production code to get it to pass. You’ll quickly get hands-on experience, writing tests for a database-driven application with the TDD methodology. Use this book to build the skills and habits that make testing a regular part of your workflow.

Who this book is written for

This book is for Django developers with little or no knowledge of test-driven development or testing in general, Familiarity with the command line, setting up a Python virtual environment, and starting a Django project are assumed.

What you will learn from this book

- Codify user stories as browser-based tests to ensure their completion
- Write isolated unit tests that not only confirm your application, but also explain it
- Use the red-green-refactor TDD cycle to create and refine your code by changing tests first
- Test integrations with external APIs by testing their documentation
- Mock out calls to external services and internal functions
- Explore the basics of documentation-driven API design
- Other testing tools available in popular Python packages such as Django REST framework and VCR.py

Kevin Harvey

Test-Driven Development with Django

Develop powerful, fully-featured Django applications by writing tests first.


Free Sample
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Keeping Your Promises'
- A synopsis of the book’s content
- More information on Test-Driven Development with Django
About the Author

Kevin Harvey first fell in love with Django while living in Quelimane, Mozambique, in 2007. His professional interests include software quality, open source, and teaching. He continues to be amazed at the Python community's ability to turn a history major into a software engineer, a feat for which he will forever be indebted. When not writing unit tests, Kevin enjoys playing the bass (both electric and stand up), and cooking with entirely too much butter. He lives in Nashville, Tennessee, with his wife and their two sons.
Writing software is hard. Even the smallest projects have many moving parts. We developers are not only expected to get those parts moving, but keep them moving as the application changes over time. Test-Driven Development (TDD) is a methodology that allows us to quantify the successful function of each of these parts before we attempt to code them. Using TDD, we can focus on a single part of the application at a time, leaving a trail of tests that guard against regression as we continue to update the application.

Django is a popular web framework written in Python. Its batteries-included: the framework itself includes URL routing, object-relational mapping, templates, and many other necessities for building a modern web application. This book will take you through the process of developing a Django app by writing failing tests first, then writing application code to make those tests pass.

What this book covers

Chapter 1, *Keeping Your Promises*, describes the benefits of TDD in comparison to other styles of programming. We'll look at a very simple example, and talk about testing as a pillar of professional software development.

Chapter 2, *Your First Test-Driven Application*, introduces the example application that we'll be building throughout the book. We'll translate a user story into a browser-based functional test using Selenium, and write unit tests and application code to start to fulfill that user story.

Chapter 3, *Ironclad Code*, continues where the previous chapter left off, digging deeper into the API available for writing unit tests in Django. We'll cover the Python Debugger, RequestFactory, and TestClient, among other tools.
Chapter 4, Building Out and Refactoring, adds new features to the application. We'll use our test suite to maintain existing functionality while refactoring to keep our code tidy and maintainable.

Chapter 5, User Stories As Code, focuses on LiveServerTestCase and the Python Selenium bindings we use to drive the browser during a test run. We'll learn how to select and click on elements, submit forms, switch between open windows, and perform other user actions in our UI.

Chapter 6, No App Is an Island, applies the TDD methodology to third-party API integration. We'll learn when, why, and how to mock out HTTP requests inside a single unit test so that our tests aren't relying on an outside resource (even if our app is).

Chapter 7, Share and Share Alike, introduces Django REST Framework—a tool for building a REST API with Django. We'll cover the importance of documentation when writing an API, and use the framework's tools to send requests to the API during tests.

Chapter 8, Promises Kept, takes a look back at what we've learned, and whether we've realized all the benefits from the first chapter. We'll get suggestions for next steps in TDD, and talk about some of the common pitfalls you may encounter.
In this chapter, we'll be introduced to Test-Driven Development. We will explore:

- What is Test-Driven Development?
- How does Test-Driven Development help build better software?
- Doesn't it take longer?
- Can't I just write the tests later?

What is Test-Driven Development?

Test-Driven Development (TDD) is the practice of:

1. writing a deliberately failing test,
2. writing application code to make the test pass,
3. refactoring to optimize the code while the test continues to pass, and
4. repeating the process until your project is complete.

The initial test is the bar you set for the logic that you want to write. It's a great way to ensure that your tests cover every nook and cranny of your code, and that it delivers exactly what you said it would.

Throughout this book, we'll explore TDD through numerous examples in a medium-sized Django project. We'll use lots of different Python utilities and see lots of sample code. The takeaway should not be any particular package (there are many other tools besides the ones we'll feature in this book), but the process itself and the change in approach required. It's a methodology, not a technology. It's a way of building applications and a discipline that requires practice.
A simple example

Here's a quick example using Python's built-in `assert`, a statement that evaluates a condition. It will throw an `AssertionError` if the condition is false, and returns nothing otherwise.

Let's say we wanted a Python function that could multiply two numbers together and return the result. Let's call it `multiplicator`.

The first step in TDD, before writing any code, is to find a way to test the application you want to write. If you're having trouble coming up with a test scenario, imagine that you've already written the application (in this case, that single function) and want to try it out in the command line. You'd probably do something like this:

```
$ python
>>> from multiplicator import multiplicator
>>> multiplicator(2, 3)
6
```

You, the human, would look at the output of the function call (6) and confirm that the operation was performed successfully. How can we teach our application to do this confirmation itself? Enter `assert`. Create a file called `multiplicator.py` and enter the following code:

```python
# multiplicator.py
assert multiplicator(2, 3) == 6
```

We can translate this statement into English as "run `multiplicator` with arguments 2 and 3 and throw an error if the returned value does not equal 6."

We now have a runnable test for our function, without so much as an attempt to write the function itself. Let's run it and see what happens:

```
$ python multiplicator.py
Traceback (most recent call last):
  File "multiplicator.py", line 1, in <module>
    assert multiplicator(2, 3) == 6
NameError: name 'multiplicator' is not defined
```

We'll get into the more interesting tools available in the `unittest` library in *Chapter 2, Your First Test-Driven Application*. For now, this is all we need to see how TDD works.
Looks like Python can't find anything called `multiplicator`. We can fix that with the following code:

```python
# multiplicator.py
def multiplicator():
    pass

assert multiplicator(2, 3) == 6
```

Try running the test now:

```
$ python multiplicator.py
```

```
Traceback (most recent call last):
  File "multiplicator.py", line 4, in <module>
    assert multiplicator(2, 3) == 6
TypeError: multiplicator() takes 0 positional arguments but 2 were given
```

Okay, our function needs to accept some arguments. Let's update it:

```python
# multiplicator.py
def multiplicator(x, y):
    pass

assert multiplicator(2, 3) == 6
```

And finally, when we run our test again:

```
$ python multiplicator.py
```

```
Traceback (most recent call last):
  File "multiplicator.py", line 4, in <module>
    assert multiplicator(2, 3) == 6
AssertionError
```

This `AssertionError` is the one we asked our test to throw (via `assert`) if the result of our function did not equal the expected value (6). Now that we're here, we can write some logic:

```python
# multiplicator.py
def multiplicator(x, y):
    i = 0
    result = 0
    while i < x:
        result += y
        i += 1
```

[3]
return result

assert multiplicator(2, 3) == 6

Whoa there, Tex! That's one way to do it... I suppose. Should we run the tests?

$ python multiplicator.py

Huh? No output? No error? You mean that monstrosity actually made the test pass?

Yes it did! We wrote application code to make the test pass without any pressure to optimize it, or without picking *the best* Python function to make it work. Now that we have the test passing, we can optimize to our heart's content; we know we're safe as long as the test continues to pass.

Let's *update* `multiplicator` to use some of Python's own integer arithmetic:

```python
# multiplicator.py
def multiplicator(x, y):
    return x*y

assert multiplicator(2, 3) == 6
```

Now, we can run our test again:

$ python multiplicator.py

That's better. We built a working, optimized function and a test suite to check for regressions using basic TDD methodology. This process is often referred to as "red/green/refactor".

**Red/green/refactor**

The smallest cycle of TDD typically involves three steps:

1. Writing a test that fails (red).
2. Doing whatever is necessary to get that test to pass (green).
3. Optimizing to fix any subpar code you may have introduced to get the test to pass (refactor).

In the preceding example, we wrote a test for the desired functionality and watched it fail (red). Then we wrote some less-than-optimal code to get it to pass (green). Finally, we refactored the code to simplify the logic while keeping the test passing (refactor). This virtuous circle is how TDD helps you write code that is both functional and beautiful.
Testing is a pillar of professional software development
There are four key practices to writing great code:

1. Version control
2. Documentation
3. Testing
4. Continuous Integration

Each builds upon the next and a thoughtful execution of each guarantees the delivery of quality software.

Version control
Version control is the ultimate undo button. It allows you to check code changes into a repository at regular intervals and rollback to any of those changes later. We’ll be using Git throughout the course of this book. To get a good primer on Git, check out http://git-scm.com/doc.

Documentation
If we're using TDD to keep promises, documentation is where we first make these promises. Simply put, documentation describes how your application works. At a minimum, your software project needs the development documentation for the next person to maintain it (even if it's you, you'll forget what you wrote). You'll probably need some less technical documentation for the end user as well.

Testing
Testing and documentation have a crucial relationship—your tests prove that your documentation is telling the truth. For instance, the documentation for a REST API may instruct a developer to send a POST request to a given URL, with a certain JSON payload in order to get back a certain response. You can ensure this is what happens by exercising this specific behavior in your tests.
Continuous Integration

All of these glorious tests will be pretty useless if no one is running them. Luckily, actually running the tests (and alerting us of any failures) is another thing we can train a machine to do. A Continuous Integration (CI) server, for our purposes, can pull our project from version control, build it, run our tests, and alert us if any errors or failures occur. It can also be the first place where our tests are run in a production-like environment (for instance, in the same operating system and database configuration), allowing us to keep our local environments configured for speed and ease.

How does TDD help in building better software?

From the outset, Test-Driven Development seems like a lot more work. We could very well be doubling the size of our code base with a test for every single branch in our logic. Here's why all that extra code will be worth it:

- **It will keep you on track**: Writing the tests first is like keeping an executable checklist of all the development tasks you have to complete. Good functional tests are the key link between user stories (which is what everyone really cares about) and your code. A well-designed functional test will ensure that the end user will be able to do everything they need to do with your application.

- **You will build exactly (and only) what is required**: As we'll see in Chapter 2, Your First Test-Driven Application, a good first step in Test-Driven Development is the translation of a user story into a distinct, self-contained functional test. Codifying the project's requirements as a test and *only writing enough code to make that test pass* will ensure that you've fulfilled all the user stories and guard against any scope creep. The project itself will help you determine when development is complete, or if any changes introduced later would interfere with any end-user functionality.

- **You're teaching your application to check itself**: Humans are better at computers in lots of ways, but the silicon has us beat when it comes to proofreading code. All we have to do is teach the machines what to look for by writing tests. Then, we can send them scampering through our files, confirming every function output, and checking every attribute of every class, any time we want.
• **It will help clarify your thinking:** Computer applications are abstract models of real-world systems that solve problems for human beings. Abstracing solutions to human problems in computer code takes serious thought and care. By clearly defining the functionality of your application with a test before you try to develop it, you force yourself to program with the end goal in mind. Having laid out the meaning of the application in a functional test (even if it's just stubbed out) helps to keep you on target even when you're elbow-deep in the logic.

• **Post-development tests just don't have the same weight:** If you try to write a test for some code that already does what you want, you'd have already closed your mind to the other possibilities of that code. You'll wind up with a narrow test that only covers that aspect of the code that you were thinking about while you were writing it. Writing the test when you're free of any preconceptions will yield a test that's more comprehensive, which will in turn produce stronger, less buggy code.

• **You will achieve flow:** TDD is all about incremental improvement. Each new test that passes (or incremental step to get to the next error in a test) is a little win. Plus, you won't have to spend hours debugging if you mess something up and a test fails. You'll be able to go right to the problem because the test that you wrote before you built that part of the application will be the one that failed.

Have you ever worked on a project where considerable effort went into maintaining a "development" database? Maybe it was set up so that you could check the effect of a custom save method from time to time? Or maybe you needed to dive into ./manage.py shell, import a bunch of your code, instantiate a few models, and then run your method to see if it worked? There's no monkey business like this when you write the tests first. The application state that you need is codified in your test suite. All that set up will happen in one command and on every run (not just when you're futzing with that method).

• **No one will ever know how buggy your code started out:** If you've worked on software projects of any complexity, you've probably developed a healthy fear of change. Change breaks stuff, particularly change to a part of an application that finds itself imported all over your project. However, if you've developed the entire application writing tests first, you've left a trail of test coverage that will alert you well before that bug you just wrote gets in to source control, let alone deployed. TDD allows you to refactor and optimize without fear of regression.
Keeping Your Promises

- **Bugs will stay fixed**: If I write a failing test that demonstrates a bug report that I receive, then update my application to make the test pass, I'll never have to worry about that bug coming back ever again because my test will catch it. Less time worrying about my production application means more fearless feature development.

- **You'll work better with your team**: An important part of working in a development team is explaining the code you write to your fellow developers. There's no better way to explain your code than to walk through your tests. Better yet, write tests as a team to foster collaboration and keep everyone on the same page.

- **You'll write testable code**: Code that is easily tested is better code. It seems both silly and obvious but it's worth mentioning. If you can prove beyond a shadow of a doubt that your code has the desired effect or return value, you'll be better able to maintain it. Writing the test before you write the code will force you to write code that can be easily tested.

- **You'll achieve the impossible**: There is nothing like a blank-slate TDD project to make you feel like you can save the world. When there is not even a hint of a function yet, you can assert any return value or effect you can imagine with any input you want. Don't hold back just because you have no idea how to build a function that would satisfy the pie-in-the-sky test you wrote. Write the test, hack away until you get it to pass, and then clean up your mess with a refactor.

![Code Snippet](image)
• **You'll be able to take big risks:** We've all been there—late in the development process or even after shipment, we see a tweak that we'd like to make in a linchpin model or method. The tweak would be a tremendous boon to system performance, but the change would have an unknown effect on nearly every other part of the application. If we've followed TDD, we'll have a complete test suite that will allow us to know the ramifications of that change immediately. We'll be able to make the change, run the tests, and see early on what it would take to keep the rest of the system in place.

• **You'll look like a pro:** When you release your code out into the world, either as a user-facing application or an installable package for other developers, you're making a promise to the people that use it. You're promising that the documentation was in fact accurate and that the dang thing does what it's supposed to do. A comprehensive test suite helps keep that promise and there's no better way to build one than by following the TDD mantra.

Particularly in the open source world, the presence of a test suite lets the community know that you're serious. It's the first thing you should look for when evaluating a new PyPI package to install. A test suite says that you can trust this software.

**Doesn't it take longer?**

A common criticism of TDD is that it slows down the development cycle. All these tests are a bunch more code. Wouldn't you have to go back and update them if you changed your application?

The answer is yes, in the short term, TDD will add time to the development cycle, particularly when you're first learning it. Writing tests is a skill and skills take practice. Once you're through the learning curve, writing test functions is much easier and faster than writing the application code. Tests are generally terse (do this, do this, check that, and so on) without complicated logic or looping. The best tests are the simplest ones. You'll be able to crank them out quickly.

The extra effort in TDD comes with the added thinking you have to do. Writing a test before you write code requires a true understanding of what you're trying to accomplish, which can be hard. But does that honestly sound like a bad thing? I'd argue that this is a decidedly positive aspect of TDD—added time spent thinking through the meaning of your code yields higher quality code. You'll uncover unforeseen complications as your tests reveal edge cases that didn't come out in code review sessions. Conversations with project owners will be more meaningful after you've put the requirements through their paces. Your application code will benefit from the extra care.
Now let's talk about the long term. Towards the end of the project, or even after launch, a big change will come down from the product owner (this is Agile, right?) or you'll find something fundamental that you want to modify. The comprehensive test suite you've built through TDD will pay you back in spades when something goes wrong, or if you need to refactor. The flexibility provided by your test suite will likely save you more time than you spent creating it. You'll thank TDD in the end.

**Can't I just write the tests later?**

There are many reasons that you may want to develop without writing tests first. Maybe you're using a new API and can't begin to think about how to write tests. Maybe you want to build a simple application quickly as a proof of concept for a client.

By all means, write code without tests, but know that code without tests is a prototype at best. Resist the urge to start the production version of your project from a testless prototype. After prototyping, start again with TDD instead of trying to go back, and write tests for the prototype.

Even if you are only creating a prototype, consider TDD for any complexity at all. If you find yourself repeatedly dropping into `./manage.py shell` sessions to set up, execute, and evaluate a function under development, write a test or two to turn that process into a single command.

**Summary**

In this chapter, we introduced the practice of TDD and the benefits of using it. In the next chapter, we will start a Django project from scratch using rigorous TDD methodology, learning some of the testing tools available in Django and Python along the way.
Where to buy this book

You can buy Test-Driven Development with Django from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.