Extracted from:

Python Testing with pytest

Simple, Rapid, Effective, and Scalable

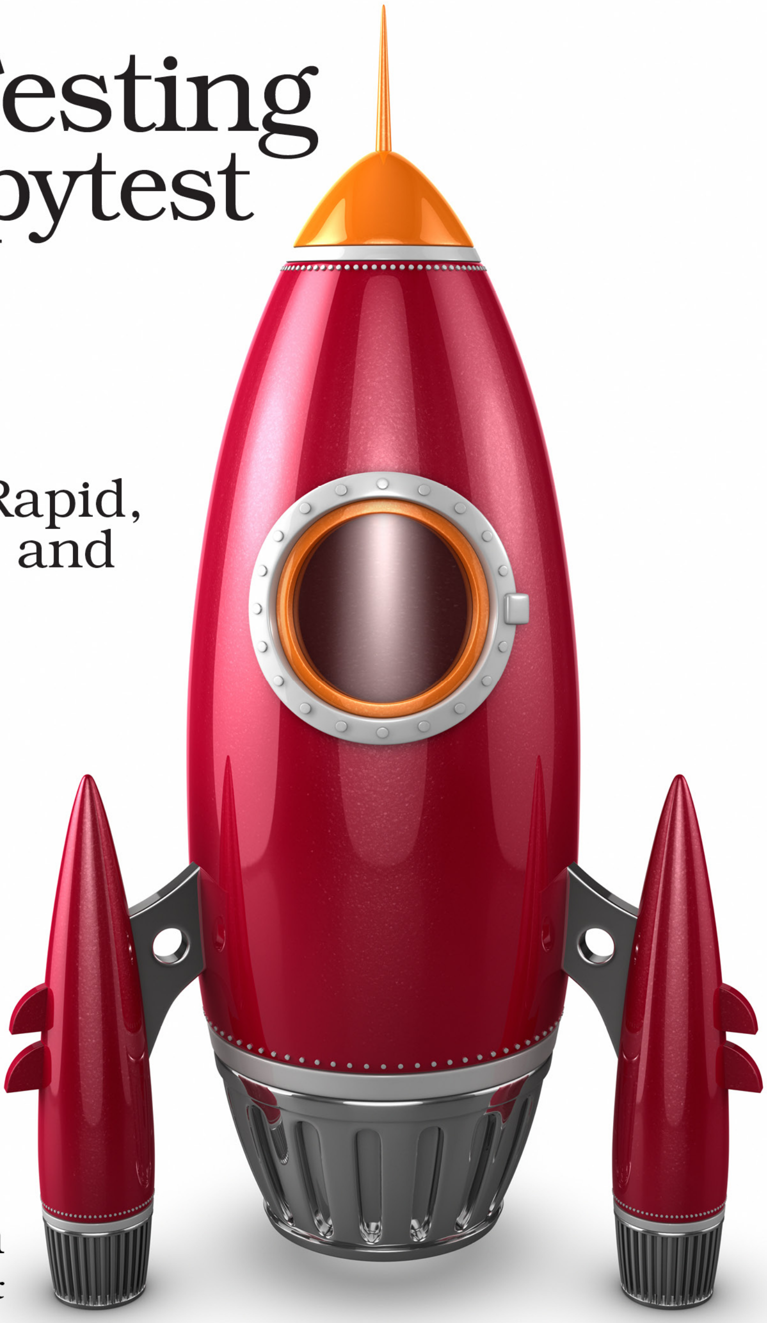
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Python Testing with pytest

Simple, Rapid, Effective, and Scalable



Brian Okken

edited by Katharine Dvorak

Python Testing with pytest

Simple, Rapid, Effective, and Scalable

Brian Okken

The Pragmatic Bookshelf

Raleigh, North Carolina

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Development Editor: Katharine Dvorak
Indexing: Potomac Indexing, LLC
Copy Editor: Nicole Abramowitz
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-240-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—September 2017

Acknowledgments

I first need to thank Michelle, my wife and best friend. I wish you could see the room I get to write in. In place of a desk, I have an antique square oak dining table to give me plenty of room to spread out papers. There's a beautiful glass-front bookcase with my retro space toys that we've collected over the years, as well as technical books, circuit boards, and juggle balls. Vintage aluminum paper storage bins are stacked on top with places for notes, cords, and even leftover book-promotion rocket stickers. One wall is covered in some velvet that we purchased years ago when a fabric store was going out of business. The fabric is to quiet the echoes when I'm recording the podcasts. I love writing here not just because it's wonderful and reflects my personality, but because it's a space that Michelle created with me and for me. She and I have always been a team, and she has been incredibly supportive of my crazy ideas to write a blog, start a podcast or two, and now, for the last year or so, write this book. She has made sure I've had time and space for writing. When I'm tired and don't think I have the energy to write, she tells me to just write for twenty minutes and see how I feel then, just like she did when she helped me get through late nights of study in college. I really, really couldn't do this without her.

I also have two amazingly awesome, curious, and brilliant daughters, Gabriella and Sophia, who are two of my biggest fans. Ella tells anyone talking about programming that they should listen to my podcasts, and Phia sported a Test & Code sticker on the backpack she took to second grade.

There are so many more people to thank.

My editor, Katharine Dvorak, helped me shape lots of random ideas and topics into a cohesive progression, and is the reason why this is a book and not a series of blog posts stapled together. I entered this project as a blogger, and a little too attached to lots of headings, subheadings, and bullet points, and Katie patiently guided me to be a better writer.

Thank you to Susannah Davidson Pfalzer, Andy Hunt, and the rest of The Pragmatic Bookshelf for taking a chance on me.

The technical reviewers have kept me honest on pytest, but also on Python style, and are the reason why the code examples are PEP 8–compliant. Thank you to Oliver Bestwalter, Florian Bruhin, Floris Bruynooghe, Mark Goody, Peter Hampton, Dave Hunt, Al Krinker, Lokesh Kumar Makani, Bruno Oliveira, Ronny Pfannschmidt, Raphael Pierzina, Luciano Ramalho, Frank Ruiz, and Dmitry Zinoviev. Many on that list are also pytest core developers and/or maintainers of incredible pytest plugins.

I need to call out Luciano for a special thank you. Partway through the writing of this book, the first four chapters were sent to a handful of reviewers. Luciano was one of them, and his review was the hardest to read. I don't think I followed all of his advice, but because of his feedback, I re-examined and rewrote much of the first three chapters and changed the way I thought about the rest of the book.

Thank you to the entire pytest-dev team for creating such a cool testing tool. Thank you to Oliver Bestwalter, Florian Bruhin, Floris Bruynooghe, Dave Hunt, Holger Krekel, Bruno Oliveira, Ronny Pfannschmidt, Raphael Pierzina, and many others for answering my pytest questions over the years.

Last but not least, I need to thank the people who have thanked me. Occasionally people email to let me know how what I've written saved them time and made their jobs easier. That's awesome, and pleases me to no end. Thank you.

Brian Okken

September 2017

Preface

The use of Python is increasing not only in software development, but also in fields such as data analysis, research science, test and measurement, and other industries. The growth of Python in many critical fields also comes with the desire to properly, effectively, and efficiently put software tests in place to make sure the programs run correctly and produce the correct results. In addition, more and more software projects are embracing continuous integration and including an automated testing phase, as release cycles are shortening and thorough manual testing of increasingly complex projects is just infeasible. Teams need to be able to trust the tests being run by the continuous integration servers to tell them if they can trust their software enough to release it.

Enter pytest.

What Is pytest?

A robust Python testing tool, pytest can be used for all types and levels of software testing. pytest can be used by development teams, QA teams, independent testing groups, individuals practicing TDD, and open source projects. In fact, projects all over the Internet have switched from unittest or nose to pytest, including Mozilla and Dropbox. Why? Because pytest offers powerful features such as ‘assert‘ rewriting, a third-party plugin model, and a powerful yet simple fixture model that is unmatched in any other testing framework.

pytest is a software test framework, which means pytest is a command-line tool that automatically finds tests you’ve written, runs the tests, and reports the results. It has a library of goodies that you can use in your tests to help you test more effectively. It can be extended by writing plugins or installing third-party plugins. It can be used to test Python distributions. And it integrates easily with other tools like continuous integration and web automation.

Here are a few of the reasons pytest stands out above many other test frameworks:

- Simple tests are simple to write in pytest.
- Complex tests are still simple to write.
- Tests are easy to read.
- Tests are easy to read. (So important it's listed twice.)
- You can get started in seconds.
- You use `assert` to fail a test, not things like `self.assertEqual()` or `self.assertLessThan()`. Just `assert`.
- You can use pytest to run tests written for unittest or nose.

pytest is being actively developed and maintained by a passionate and growing community. It's so extensible and flexible that it will easily fit into your work flow. And because it's installed separately from your Python version, you can use the same latest version of pytest on legacy Python 2 (2.6 and above) and Python 3 (3.3 and above).

Learn pytest While Testing an Example Application

How would you like to learn pytest by testing silly examples you'd never run across in real life? Me neither. We're not going to do that in this book. Instead, we're going to write tests against an example project that I hope has many of the same traits of applications you'll be testing after you read this book.

The Tasks Project

The application we'll look at is called Tasks. Tasks is a minimal task-tracking application with a command-line user interface. It has enough in common with many other types of applications that I hope you can easily see how the testing concepts you learn while developing tests against Tasks are applicable to your projects now and in the future.

While Tasks has a command-line interface (CLI), the CLI interacts with the rest of the code through an application programming interface (API). The API is the interface where we'll direct most of our testing. The API interacts with a database control layer, which interacts with a document database—either MongoDB or TinyDB. The type of database is configured at database initialization.

Before we focus on the API, let's look at `tasks`, the command-line tool that represents the user interface for Tasks.

Here's an example session:

```
$ tasks add 'do something' --owner Brian
$ tasks add 'do something else'
$ tasks list
  ID      owner  done summary
  --      -----  ---- -------
   1      Brian False do something
   2            False do something else
$ tasks update 2 --owner Brian
$ tasks list
  ID      owner  done summary
  --      -----  ---- -------
   1      Brian False do something
   2      Brian False do something else
$ tasks update 1 --done True
$ tasks list
  ID      owner  done summary
  --      -----  ---- -------
   1      Brian  True do something
   2      Brian False do something else
$ tasks delete 1
$ tasks list
  ID      owner  done summary
  --      -----  ---- -------
   2      Brian False do something else
$
```

This isn't the most sophisticated task-management application, but it's complicated enough to use it to explore testing.

Test Strategy

While pytest is useful for unit testing, integration testing, system or end-to-end testing, and functional testing, the strategy for testing the Tasks project focuses primarily on subcutaneous functional testing. Following are some helpful definitions:

- *Unit test*: A test that checks a small bit of code, like a function or a class, in isolation of the rest of the system. I consider the tests in Chapter 1, *Getting Started with pytest*, on page ?, to be unit tests run against the Tasks data structure.
- *Integration test*: A test that checks a larger bit of the code, maybe several classes, or a subsystem. Mostly it's a label used for some test larger than a unit test, but smaller than a system test.
- *System test (end-to-end)*: A test that checks all of the system under test in an environment as close to the end-user environment as possible.

- *Functional test:* A test that checks a single bit of functionality of a system. A test that checks how well we add or delete or update a task item in Tasks is a functional test.
- *Subcutaneous test*: A test that doesn't run against the final end-user interface, but against an interface just below the surface. Since most of the tests in this book test against the API layer—not the CLI—they qualify as subcutaneous tests.

How This Book Is Organized

In Chapter 1, *Getting Started with pytest,* on page ?, you'll install pytest and get it ready to use. You'll then take one piece of the Tasks project—the data structure representing a single task (a namedtuple called Task)—and use it to test examples. You'll learn how to run pytest with a handful of test files. You'll look at many of the popular and hugely useful command-line options for pytest, such as being able to re-run test failures, stop execution after the first failure, control the stack trace and test run verbosity, and much more.

In Chapter 2, *Writing Test Functions,* on page ?, you'll install Tasks locally using pip and look at how to structure tests within a Python project. You'll do this so that you can get to writing tests against a real application. All the examples in this chapter run tests against the installed application, including writing to the database. The actual test functions are the focus of this chapter, and you'll learn how to use assert effectively in your tests. You'll also learn about markers, a feature that allows you to mark many tests to be run at one time, mark tests to be skipped, or tell pytest that we already know some tests will fail. And I'll cover how to run just some of the tests, not just with markers, but by structuring our test code into directories, modules, and classes, and how to run these subsets of tests.

Not all of your test code goes into test functions. In Chapter 3, *pytest Fixtures,* on page ?, you'll learn how to put test data into test fixtures, as well as set up and tear down code. Setting up system state (or subsystem or unit state) is an important part of software testing. You'll explore this aspect of pytest fixtures to help get the Tasks project's database initialized and prefilled with test data for some tests. Fixtures are an incredibly powerful part of pytest, and you'll learn how to use them effectively to further reduce test code duplication and help make your test code incredibly readable and maintainable. pytest fixtures are also parametrizable, similar to test functions, and you'll use this feature to be able to run all of your tests against both TinyDB and MongoDB, the database back ends supported by Tasks.

In Chapter 4, *Builtin Fixtures*, on page ?, you will look at some builtin fixtures provided out-of-the-box by pytest. You will learn how pytest builtin fixtures can keep track of temporary directories and files for you, help you test output from your code under test, use monkey patches, check for warnings, and more.

In Chapter 5, *Plugins*, on page ?, you'll learn how to add command-line options to pytest, alter the pytest output, and share pytest customizations, including fixtures, with others through writing, packaging, and distributing your own plugins. The plugin we develop in this chapter is used to make the test failures we see while testing Tasks just a little bit nicer. You'll also look at how to properly test your test plugins. How's that for meta? And just in case you're not inspired enough by this chapter to write some plugins of your own, I've hand-picked a bunch of great plugins to show off what's possible in Appendix 3, *Plugin Sampler Pack*, on page ?.

Speaking of customization, in Chapter 6, *Configuration*, on page ?, you'll learn how you can customize how pytest runs by default for your project with configuration files. With a `pytest.ini` file, you can do things like store command-line options so you don't have to type them all the time, tell pytest to not look into certain directories for test files, specify a minimum pytest version your tests are written for, and more. These configuration elements can be put in `tox.ini` or `setup.cfg` as well.

In the final chapter, Chapter 7, *Using pytest with Other Tools*, on page ?, you'll look at how you can take the already powerful pytest and supercharge your testing with complementary tools. You'll run the Tasks project on multiple versions of Python with tox. You'll test the Tasks CLI while not having to run the rest of the system with mock. You'll use `coverage.py` to see if any of the Tasks project source code isn't being tested. You'll use Jenkins to run test suites and display results over time. And finally, you'll see how pytest can be used to run unittest tests, as well as share pytest style fixtures with unittest-based tests.

What You Need to Know

Python
: You don't need to know a lot of Python. The examples don't do anything super weird or fancy.

pip
: You should use `pip` to install pytest and pytest plugins. If you want a refresher on `pip`, check out Appendix 2, *pip*, on page ?.

A command line

I wrote this book and captured the example output using bash on a Mac laptop. However, the only commands I use in bash are `cd` to go to a specific directory, and `pytest`, of course. Since `cd` exists in Windows `cmd.exe` and all unix shells that I know of, all examples should be runnable on whatever terminal-like application you choose to use.

That's it, really. You don't need to be a programming expert to start writing automated software tests with pytest.

Example Code and Online Resources

The examples in this book were written using Python 3.6 and pytest 3.2. pytest 3.2 supports Python 2.6, 2.7, and Python 3.3+.

The source code for the Tasks project, as well as for all of the tests shown in this book, is available through a link[1] on the book's web page at pragprog.com.[2] You don't need to download the source code to understand the test code; the test code is presented in usable form in the examples. But to follow along with the Tasks project, or to adapt the testing examples to test your own project (more power to you!), you must go to the book's web page to download the Tasks project. Also available on the book's web page is a link to post errata[3] and a discussion forum.[4]

I've been programming for over twenty-five years, and nothing has made me love writing test code as much as pytest. I hope you learn a lot from this book, and I hope that you'll end up loving test code as much as I do.

1. https://pragprog.com/titles/bopytest/source_code
2. https://pragprog.com/titles/bopytest
3. https://pragprog.com/titles/bopytest/errata
4. https://forums.pragprog.com/forums/438