Extracted from:

Python Testing with pytest

Simple, Rapid, Effective, and Scalable

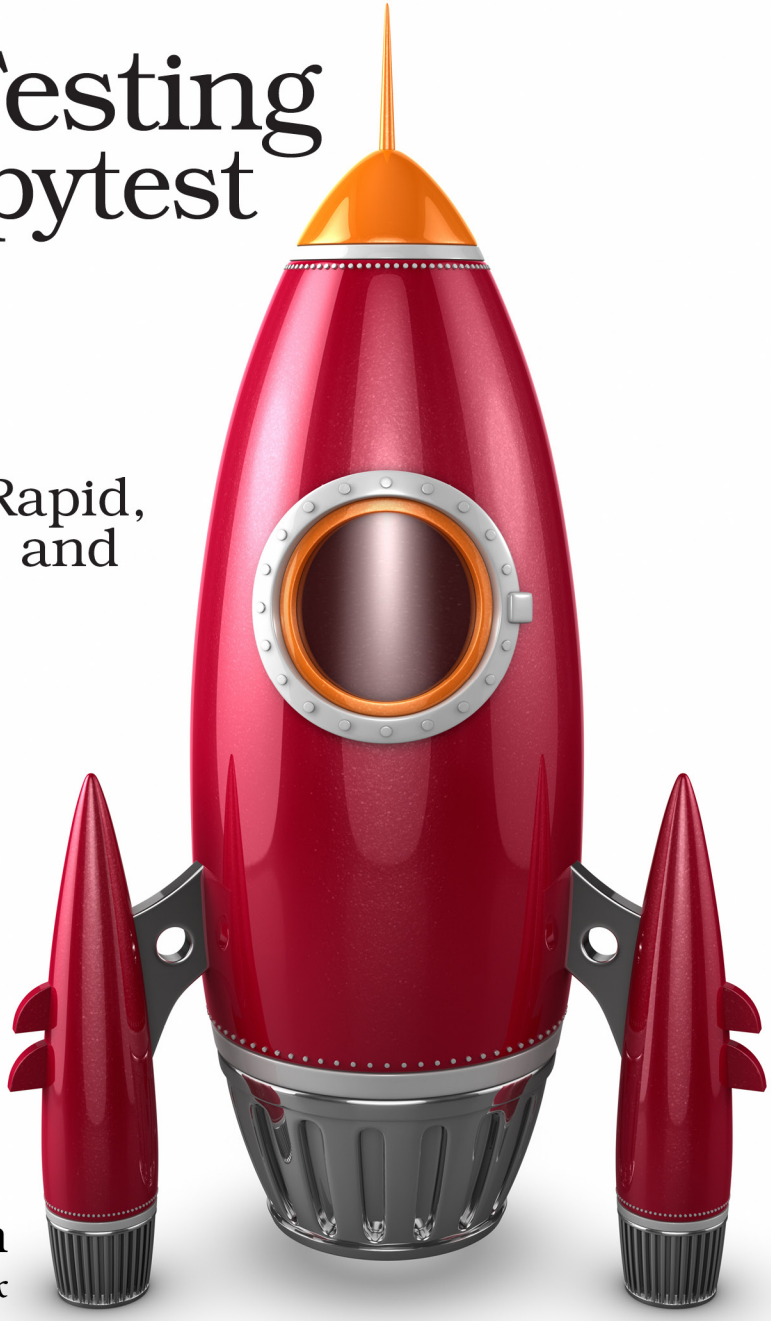
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Python Testing with pytest

Simple, Rapid, Effective, and Scalable



Brian Okken

edited by Katharine Dvorak

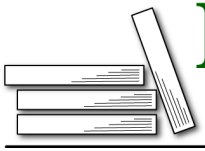
Python Testing with pytest

Simple, Rapid, Effective, and Scalable

Brian Okken

The Pragmatic Bookshelf

Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

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: B1.0—May 17, 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 it 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(), self.assertLessThan(), etc. 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 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. We'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. We'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. We'll do this so that we 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 affectively 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 we'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 setup and teardown code. Setting up system state (or subsystem or unit state) is an important part of software testing. We explore this aspect of pytest fixtures to help us with the Tasks project in getting the database initialized and pre-filling it 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 parameterizable, similar to test functions, and we use this feature to be able to run all of our tests against both TinyDB and MongoDB, the database backends supported by Tasks.

In Chapter 4, *Builtin Fixtures*, on page ?, we'll look at some builtin fixtures provided out-of-the-box by pytest. You'll learn how pytest builtin fixtures can keep track of temporary directories and files for you, can 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. We also look at how to properly test our 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 ?, we'll look at how we can take the already powerful pytest and supercharge our testing with some 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* to see if any of the Tasks source code isn't being tested. You'll use pytest to run legacy unittest tests, and check the docstring code examples with doctest.

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 code for the Tasks project, as well as all of the tests shown in this book, are available through a link[1] on the book's webpage at pragprog.com.[2] You don't need to download the source code in order 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 webpage to download the Tasks project. Also available on the book's webpage is a link to post errata and a discussion forum.

I've been programming for over 25 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