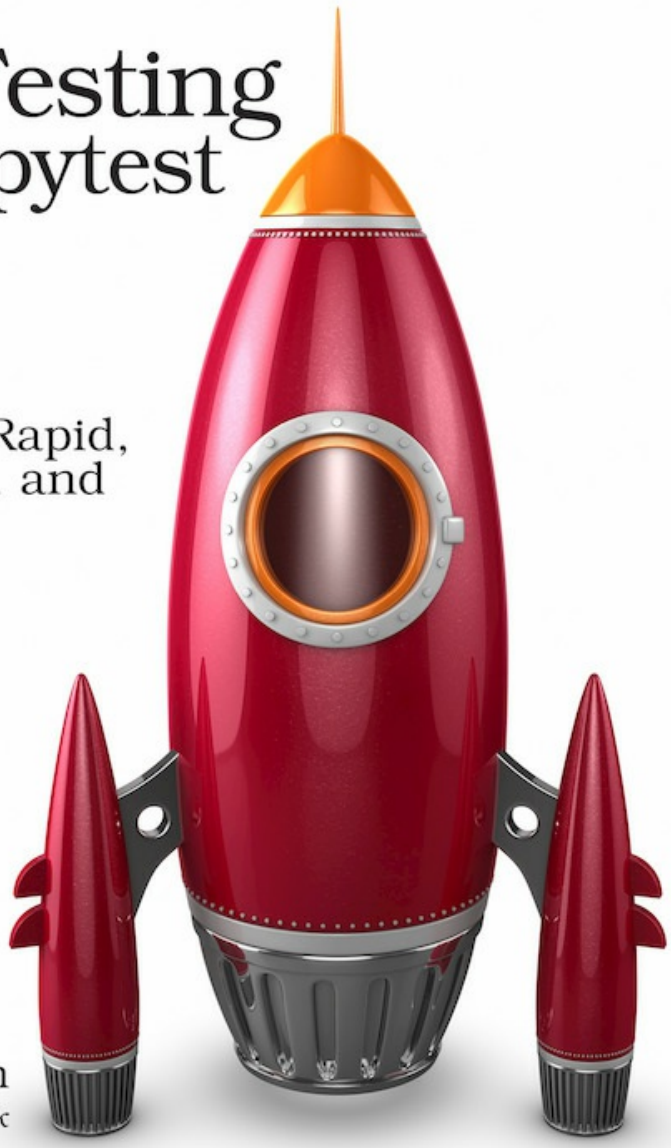
The Pragmatic Programmers

Python Testing with pytest

Simple, Rapid, Effective, and Scalable



Brian Okken

edited by Katharine Dvorak

Python Testing with pytest

Simple, Rapid, Effective, and Scalable

by Brian Okken

Version: P1.0 (September 2017)

About the Pragmatic Bookshelf

The Pragmatic Bookshelf is an agile publishing company. We're here because we want to improve the lives of developers. We do this by creating timely, practical titles, written by programmers for programmers.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at http://pragprog.com.

Our ebooks do not contain any Digital Restrictions Management, and have always been DRM-free. We pioneered the beta book concept, where you can purchase and read a book while it's still being written, and provide feedback to the author to help make a better book for everyone. Free resources for all purchasers include source code downloads (if applicable), errata and discussion forums, all available on the book's home page at pragprog.com. We're here to make your life easier.

New Book Announcements

Want to keep up on our latest titles and announcements, and occasional special offers? Just create an account on pragprog.com (an email address and a password is all it takes) and select the checkbox to receive newsletters. You can also follow us on twitter as @pragprog.

About Ebook Formats

If you buy directly from pragprog.com, you get ebooks in all available formats for one price. You can synch your ebooks amongst all your devices (including iPhone/iPad, Android, laptops, etc.) via Dropbox. You get free updates for the life of the edition. And, of course, you can always come back and re-download your books when needed. Ebooks bought from the Amazon Kindle store are subject to Amazon's polices. Limitations in Amazon's file format may cause ebooks to display differently on different devices. For more information, please see our FAQ at pragprog.com/frequently-asked-questions/ebooks. To learn more about this book and access the free resources, go to https://pragprog.com/book/bopytest, the book's homepage.

Thanks for your continued support,

Andy Hunt
The Pragmatic Programmers

The team that produced this book includes: Andy Hunt (Publisher)Janet Furlow (VP of Operations)Katharine Dvorak (Development Editor)Potomac Indexing, LLC (Indexing)Nicole Abramowitz (Copy Editor)Gilson Graphics (Layout)

For customer support, please contact support@pragprog.com.

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

Table of Contents

Early praise for *Python Testing with pytest*

I found *Python Testing with pytest* to be an eminently usable introductory guidebook to the pytest testing framework. It is already paying dividends for me at my company.

→ Chris Shaver
VP of Product, Uprising Technology

Systematic software testing, especially in the Python community, is often either completely overlooked or done in an ad hoc way. Many Python programmers are completely unaware of the existence of pytest. Brian Okken takes the trouble to show that software testing with pytest is easy, natural, and even exciting.

→ Dmitry Zinoviev
Author of Data Science Essentials in Python

This book is the missing chapter absent from every comprehensive Python book.

→ Frank Ruiz
Principal Site Reliability Engineer, Box, Inc.

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*, 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*, 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*, 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*, 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*, 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*, 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*.

Speaking of customization, in Chapter 6, *Configuration*, 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*, 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*.

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.

Footnotes

[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

Chapter 1
Getting Started with pytest

This is a test:

ch1/test_one.py

```
def test_passing():
assert (1, 2, 3) == (1, 2, 3)
```

This is what it looks like when it's run:

```
$ cd /path/to/code/ch1
$ pytest test_one.py
====================== test session starts ======================
collected 1 items

test_one.py .

=================== 1 passed in 0.01 seconds ====================
```

The dot after test_one.py means that one test was run and it passed. If you need more information, you can use -v or --verbose:

```
$ pytest -v test_one.py
====================== test session starts ======================
collected 1 items

test_one.py::test_passing PASSED

=================== 1 passed in 0.01 seconds ====================
```

If you have a color terminal, the PASSED and bottom line are green. It's nice.

This is a failing test:

ch1/test_two.py

```
def test_failing():
assert (1, 2, 3) == (3, 2, 1)
```

The way pytest shows you test failures is one of the many reasons developers love pytest. Let's watch this fail:

```
$ pytest test_two.py
```

```
===================== test session starts =====================
collected 1 items

test_two.py F

============================ FAILURES ============================
_____________________________ test_failing _____________________________

def test_failing():
> assert (1, 2, 3) == (3, 2, 1)
E assert (1, 2, 3) == (3, 2, 1)
E At index 0 diff: 1 != 3
E Use -v to get the full diff

test_two.py:2: AssertionError
=================== 1 failed in 0.04 seconds ===================
```

Cool. The failing test, test_failing, gets its own section to show us why it failed. And pytest tells us exactly what the first failure is: index 0 is a mismatch. Much of this is in red to make it really stand out (if you've got a color terminal). That's already a lot of information, but there's a line that says Use -v to get the full diff. Let's do that:

```
$ pytest -v test_two.py
===================== test session starts =====================
collected 1 items

test_two.py::test_failing FAILED

============================ FAILURES ============================
_____________________________ test_failing _____________________________

def test_failing():
> assert (1, 2, 3) == (3, 2, 1)
E assert (1, 2, 3) == (3, 2, 1)
E At index 0 diff: 1 != 3
E Full diff:
E - (1, 2, 3)
E ? ^ ^
E + (3, 2, 1)
E ? ^ ^

test_two.py:2: AssertionError
=================== 1 failed in 0.04 seconds ===================
```

Wow. pytest adds little carets (^) to show us exactly what's different.

If you're already impressed with how easy it is to write, read, and run tests with pytest, and how easy it is to read the output to see where the tests fail, well, you ain't seen nothing yet. There's lots more where that came from. Stick around and let me show you why I think pytest is the absolute best test framework available.

In the rest of this chapter, you'll install pytest, look at different ways to run it, and run through some of the most often used command-line options. In future chapters, you'll learn how to write test functions that maximize the power of pytest, how to pull setup code into setup and teardown sections called fixtures, and how to use fixtures and plugins to really supercharge your software testing.

But first, I have an apology. I'm sorry that the test, assert (1, 2, 3) == (3, 2, 1), is so boring. Snore. No one would write a test like that in real life. Software tests are comprised of code that tests other software that you aren't always positive will work. And (1, 2, 3) == (1, 2, 3) will always work. That's why we won't use overly silly tests like this in the rest of the book. We'll look at tests for a real software project. We'll use an example project called Tasks that needs some test code. Hopefully it's simple enough to be easy to understand, but not so simple as to be boring.

Another great use of software tests is to test your assumptions about how the software under test works, which can include testing your understanding of third-party modules and packages, and even builtin Python data structures. The Tasks project uses a structure called Task, which is based on the namedtuple factory method, which is part of the standard library. The Task structure is used as a data structure to pass information between the UI and the API. For the rest of this chapter, I'll use Task to demonstrate running pytest and using some frequently used command-line options.

Here's Task:

```
from collections import namedtuple
Task = namedtuple('Task', ['summary', 'owner', 'done', 'id'])
```

The namedtuple() factory function has been around since Python 2.6, but I still find that many Python developers don't know how cool it is. At the very least, using Task for test examples will be more interesting than (1, 2, 3) == (1, 2, 3) or add(1, 2) == 3.

Before we jump into the examples, let's take a step back and talk about how to get pytest and install it.

Getting pytest

The headquarters for pytest is https://docs.pytest.org. That's the official documentation. But it's distributed through PyPI (the Python Package Index) at https://pypi.python.org/pypi/pytest.

Like other Python packages distributed through PyPI, use pip to install pytest into the virtual environment you're using for testing:

```
$ pip3 install -U virtualenv
$ python3 -m virtualenv venv
$ source venv/bin/activate
$ pip install pytest
```

If you are not familiar with virtualenv or pip, I have got you covered. Check out Appendix 1, *Virtual Environments* and Appendix 2, *pip*.

What About Windows, Python 2, and venv?

The example for virtualenv and pip should work on many POSIX systems, such as Linux and macOS, and many versions of Python, including Python 2.7.9 and later.

The source venv/bin/activate line won't work for Windows, use venv\Scripts\activate.bat instead. Do this:

```
C:\> pip3 install -U virtualenv
C:\> python3 -m virtualenv venv
C:\> venv\Scripts\activate.bat
(venv) C:\> pip install pytest
```

For Python 3.6 and above, you may get away with using venv instead of virtualenv, and you don't have to install it first. It's included in Python 3.6 and above. However, I've heard that some platforms still behave better with virtualenv.

Running pytest

```
$ pytest --help
usage: pytest [options] [file_or_dir] [file_or_dir] [...]
 ...
```

Given no arguments, pytest looks at your current directory and all subdirectories for test files and runs the test code it finds. If you give pytest a filename, a directory name, or a list of those, it looks there instead of the current directory. Each directory listed on the command line is recursively traversed to look for test code.

For example, let's create a subdirectory called tasks, and start with this test file:

ch1/tasks/test_three.py

```
"""Test the Task data type."""

from collections import namedtuple

Task = namedtuple('Task', ['summary', 'owner', 'done', 'id'])
Task.__new__.__defaults__ = (None, None, False, None)


def test_defaults():
"""Using no parameters should invoke defaults."""
t1 = Task()
t2 = Task(None, None, False, None)
assert t1 == t2


def test_member_access():
"""Check .field functionality of namedtuple."""
t = Task('buy milk', 'brian')
assert t.summary == 'buy milk'
assert t.owner == 'brian'
assert (t.done, t.id) == (False, None)
```

You can use __new__.__defaults__ to create Task objects without having to specify all the fields. The test_defaults() test is there to demonstrate and validate how the defaults work.

The test_member_access() test is to demonstrate how to access members by name and not by index, which is one of the main reasons to use namedtuples.

Let's put a couple more tests into a second file to demonstrate the _asdict() and _replace() functionality:

ch1/tasks/test_four.py

```
"""Test the Task data type."""

from collections import namedtuple


Task = namedtuple('Task', ['summary', 'owner', 'done', 'id'])
Task.__new__.__defaults__ = (None, None, False, None)


def test_asdict():
"""_asdict() should return a dictionary."""
t_task = Task('do something', 'okken', True, 21)
t_dict = t_task._asdict()
expected = {'summary': 'do something',
'owner': 'okken',
'done': True,
'id': 21}
assert t_dict == expected


def test_replace():
"""replace() should change passed in fields."""
t_before = Task('finish book', 'brian', False)
t_after = t_before._replace(id=10, done=True)
t_expected = Task('finish book', 'brian', True, 10)
assert t_after == t_expected
```

To run pytest, you have the option to specify files and directories. If you don't specify any files or directories, pytest will look for tests in the current working directory and subdirectories. It looks for files starting with test_ or ending with _test. From the ch1 directory, if you run pytest with no commands, you'll run four files' worth of tests:

```
$ cd /path/to/code/ch1
$ pytest
====================== test session starts ======================
collected 6 items

test_one.py .
test_two.py F
tasks/test_four.py ..
tasks/test_three.py ..
```

```
============================ FAILURES ============================
_________________________ test_failing _________________________

def test_failing():
> assert (1, 2, 3) == (3, 2, 1)
E assert (1, 2, 3) == (3, 2, 1)
E At index 0 diff: 1 != 3
E Use -v to get the full diff

test_two.py:2: AssertionError
============== 1 failed, 5 passed in 0.08 seconds ==============
```

To get just our new task tests to run, you can give pytest all the filenames you want run, or the directory, or call pytest from the directory where our tests are:

```
$ pytest tasks/test_three.py tasks/test_four.py
====================== test session starts ======================
collected 4 items

tasks/test_three.py ..
tasks/test_four.py ..

=================== 4 passed in 0.02 seconds ====================
$ pytest tasks
====================== test session starts ======================
collected 4 items

tasks/test_four.py ..
tasks/test_three.py ..

=================== 4 passed in 0.03 seconds ====================
$ cd /path/to/code/ch1/tasks
$ pytest
====================== test session starts ======================
collected 4 items

test_four.py ..
test_three.py ..

=================== 4 passed in 0.02 seconds ====================
```

The part of pytest execution where pytest goes off and finds which tests to run is called test discovery. pytest was able to find all the tests we wanted it to run because we named them

according to the pytest naming conventions. Here's a brief overview of the naming conventions to keep your test code discoverable by pytest:

- Test files should be named test_<something>.py or <something>_test.py.
- Test methods and functions should be named test_<something>.
- Test classes should be named Test<Something>.

Since our test files and functions start with test_, we're good. There are ways to alter these discovery rules if you have a bunch of tests named differently. I'll cover that in Chapter 6, *Configuration*.

Let's take a closer look at the output of running just one file:

```
$ cd /path/to/code/ch1/tasks
$ pytest test_three.py
================= test session starts ==================
platform darwin -- Python 3.6.2, pytest-3.2.1, py-1.4.34, pluggy-0.4.0
rootdir: /path/to/code/ch1/tasks, inifile:
collected 2 items

test_three.py ..

=============== 2 passed in 0.01 seconds ===============
```

The output tells us quite a bit.

===== test session starts ====

> pytest provides a nice delimiter for the start of the test session. A session is one invocation of pytest, including all of the tests run on possibly multiple directories. This definition of session becomes important when I talk about session scope in relation to pytest fixtures in *Specifying Fixture Scope*.

platform darwin -- Python 3.6.2, pytest-3.2.1, py-1.4.34, pluggy-0.4.0

> platform darwin is a Mac thing. This is different on a Windows machine. The Python and pytest versions are listed, as well as the packages pytest depends on. Both py and pluggy are packages developed by the pytest team to help with the implementation of pytest.

rootdir: /path/to/code/ch1/tasks, inifile:

> The rootdir is the topmost common directory to all of the directories being searched for test code. The inifile (blank here) lists the configuration file being used. Configuration files could be pytest.ini, tox.ini, or setup.cfg. You'll look at configuration files in more detail in Chapter 6, *Configuration*.

collected 2 items

> These are the two test functions in the file.

test_three.py ..

> The test_three.py shows the file being tested. There is one line for each test file. The two dots denote that the tests passed—one dot for each test function or method. Dots are only for passing tests. Failures, errors, skips, xfails, and xpasses are denoted with F, E, s, x, and X, respectively. If you want to see more than dots for passing tests, use the -v or --verbose option.

== 2 passed in 0.01 seconds ==

> This refers to the number of passing tests and how long the entire test session took. If non-passing tests were present, the number of each category would be listed here as well.

The outcome of a test is the primary way the person running a test or looking at the results understands what happened in the test run. In pytest, test functions may have several different outcomes, not just pass or fail.

Here are the possible outcomes of a test function:

- PASSED (.): The test ran successfully.
- FAILED (F): The test did not run successfully (or XPASS + strict).
- SKIPPED (s): The test was skipped. You can tell pytest to skip a test by using either the @pytest.mark.skip() or pytest.mark.skipif() decorators, discussed in *Skipping Tests*.
- xfail (x): The test was not supposed to pass, ran, and failed. You can tell pytest that a test is expected to fail by using the @pytest.mark.xfail() decorator, discussed in *Marking Tests as Expecting to Fail*.
- XPASS (X): The test was not supposed to pass, ran, and passed.
- ERROR (E): An exception happened outside of the test function, in either a fixture, discussed in Chapter 3, *pytest Fixtures*, or in a hook function, discussed in Chapter 5, *Plugins*.

Running Only One Test

One of the first things you'll want to do once you've started writing tests is to run just one. Specify the file directly, and add a ::test_name, like this:

```
$ cd /path/to/code/ch1
$ pytest -v tasks/test_four.py::test_asdict
================== test session starts ==================
collected 3 items

tasks/test_four.py::test_asdict PASSED

================ 1 passed in 0.01 seconds ================
```

Now, let's take a look at some of the options.

Using Options

We've used the verbose option, -v or --verbose, a couple of times already, but there are many more options worth knowing about. We're not going to use all of the options in this book, but quite a few. You can see all of them with pytest --help.

The following are a handful of options that are quite useful when starting out with pytest. This is by no means a complete list, but these options in particular address some common early desires for controlling how pytest runs when you're first getting started.

```
$ pytest --help
 ... subset of the list ...
-k EXPRESSION only run tests/classes which match the given
substring expression.
Example: -k 'test_method or test_other' matches
all test functions and classes whose name
contains 'test_method' or 'test_other'.
-m MARKEXPR only run tests matching given mark expression.
example: -m 'mark1 and not mark2'.
-x, --exitfirst exit instantly on first error or failed test.
--maxfail=num exit after first num failures or errors.
--capture=method per-test capturing method: one of fd|sys|no.
-s shortcut for --capture=no.
--lf, --last-failed rerun only the tests that failed last time
(or all if none failed)
--ff, --failed-first run all tests but run the last failures first.
-v, --verbose increase verbosity.
-q, --quiet decrease verbosity.
-l, --showlocals show locals in tracebacks (disabled by default).
--tb=style traceback print mode (auto/long/short/line/native/no).
--durations=N show N slowest setup/test durations (N=0 for all).
--collect-only only collect tests, don't execute them.
--version display pytest lib version and import information.
-h, --help show help message and configuration info
```

–collect-only

The --collect-only option shows you which tests will be run with the given options and configuration. It's convenient to show this option first so that the output can be used as a reference for the rest of the examples. If you start in the ch1 directory, you should see all of the test functions you've looked at so far in this chapter:

```
$ cd /path/to/code/ch1
```

```
$ pytest --collect-only
================== test session starts ==================
collected 6 items
<Module 'test_one.py'>
<Function 'test_passing'>
<Module 'test_two.py'>
<Function 'test_failing'>
<Module 'tasks/test_four.py'>
<Function 'test_asdict'>
<Function 'test_replace'>
<Module 'tasks/test_three.py'>
<Function 'test_defaults'>
<Function 'test_member_access'>

============== no tests ran in 0.03 seconds ==============
```

The --collect-only option is helpful to check if other options that select tests are correct before running the tests. We'll use it again with -k to show how that works.

-k EXPRESSION

The -k option lets you use an expression to find what test functions to run. Pretty powerful. It can be used as a shortcut to running an individual test if its name is unique, or running a set of tests that have a common prefix or suffix in their names. Let's say you want to run the test_asdict() and test_defaults() tests. You can test out the filter with --collect-only:

```
$ cd /path/to/code/ch1
$ pytest -k "asdict or defaults" --collect-only
================== test session starts ==================
collected 6 items
<Module 'tasks/test_four.py'>
<Function 'test_asdict'>
<Module 'tasks/test_three.py'>
<Function 'test_defaults'>

================== 4 tests deselected ==================
============== 4 deselected in 0.03 seconds ==============
```

Yep. That looks like what we want. Now you can run them by removing the --collect-only:

```
$ pytest -k "asdict or defaults"
================== test session starts ==================
collected 6 items
```

```
tasks/test_four.py .
tasks/test_three.py .

================== 4 tests deselected ==================
========= 2 passed, 4 deselected in 0.03 seconds ==========
```

Hmm. Just dots. So they passed. But were they the right tests? One way to find out is to use -v or --verbose:

```
$ pytest -v -k "asdict or defaults"
================== test session starts ==================
collected 6 items

tasks/test_four.py::test_asdict PASSED
tasks/test_three.py::test_defaults PASSED

================== 4 tests deselected ==================
========= 2 passed, 4 deselected in 0.02 seconds ==========
```

Yep. They were the correct tests.

-m MARKEXPR

Markers are one of the best ways to mark a subset of your test functions so that they can be run together. As an example, one way to run test_replace() and test_member_access(), even though they are in separate files, is to mark them.

You can use any marker name. Let's say you want to use run_these_please. You'd mark a test using the decorator @pytest.mark.run_these_please, like so:

```
import pytest


...
@pytest.mark.run_these_please
def test_member_access():
...
```

Then you'd do the same for test_replace(). You can then run all the tests with the same marker with pytest -m run_these_please:

```
$ cd /path/to/code/ch1/tasks
$ pytest -v -m run_these_please
================== test session starts ==================
collected 4 items
```

```
test_four.py::test_replace PASSED
test_three.py::test_member_access PASSED

================== 2 tests deselected ==================
========= 2 passed, 2 deselected in 0.02 seconds =========
```

The marker expression doesn't have to be a single marker. You can say things like -m "mark1 and mark2" for tests with both markers, -m "mark1 and not mark2" for tests that have mark1 but not mark2, -m "mark1 or mark2" for tests with either, and so on. I'll discuss markers more completely in *Marking Test Functions*.

-x, –exitfirst

Normal pytest behavior is to run every test it finds. If a test function encounters a failing assert or an exception, the execution for that test stops there and the test fails. And then pytest runs the next test. Most of the time, this is what you want. However, especially when debugging a problem, stopping the entire test session immediately when a test fails is the right thing to do. That's what the -x option does.

Let's try it on the six tests we have so far:

```
$ cd /path/to/code/ch1
$ pytest -x
================== test session starts ==================
collected 6 items

test_one.py .
test_two.py F

======================== FAILURES ========================
_____________________ test_failing _____________________

def test_failing():
> assert (1, 2, 3) == (3, 2, 1)
E assert (1, 2, 3) == (3, 2, 1)
E At index 0 diff: 1 != 3
E Use -v to get the full diff

test_two.py:2: AssertionError
!!!!!!!!! Interrupted: stopping after 1 failures !!!!!!!!!!
========== 1 failed, 1 passed in 0.25 seconds ===========
```

Near the top of the output you see that all six tests (or "items") were collected, and in the bottom line you see that one test failed and one passed, and pytest displays the "Interrupted" line to tell us that it stopped.

Without -x, all six tests would have run. Let's run it again without the -x. Let's also use --tb=no to turn off the stack trace, since you've already seen it and don't need to see it again:

```
$ cd /path/to/code/ch1
$ pytest --tb=no
================== test session starts ==================
collected 6 items

test_one.py .
test_two.py F
tasks/test_four.py ..
tasks/test_three.py ..

========== 1 failed, 5 passed in 0.09 seconds ===========
```

This demonstrates that without the -x, pytest notes failure in test_two.py and continues on with further testing.

–maxfail=num

The -x option stops after one test failure. If you want to let some failures happen, but not a ton, use the --maxfail option to specify how many failures are okay with you.

It's hard to really show this with only one failing test in our system so far, but let's take a look anyway. Since there is only one failure, if we set --maxfail=2, all of the tests should run, and --maxfail=1 should act just like -x:

```
$ cd /path/to/code/ch1
$ pytest --maxfail=2 --tb=no
================== test session starts ==================
collected 6 items

test_one.py .
test_two.py F
tasks/test_four.py ..
tasks/test_three.py ..

========== 1 failed, 5 passed in 0.08 seconds ===========
$ pytest --maxfail=1 --tb=no
================== test session starts ==================
collected 6 items

test_one.py .
test_two.py F
```

!!!!!!!!! Interrupted: stopping after 1 failures !!!!!!!!!!
========== 1 failed, 1 passed in 0.19 seconds ============

Again, we used --tb=no to turn off the traceback.

-s and –capture=method

The -s flag allows print statements—or really any output that normally would be printed to stdout—to actually be printed to stdout while the tests are running. It is a shortcut for --capture=no. This makes sense once you understand that normally the output is captured on all tests. Failing tests will have the output reported after the test runs on the assumption that the output will help you understand what went wrong. The -s or --capture=no option turns off output capture. When developing tests, I find it useful to add several print() statements so that I can watch the flow of the test.

Another option that may help you to not need print statements in your code is -l/--showlocals, which prints out the local variables in a test if the test fails.

Other options for capture method are --capture=fd and --capture=sys. The --capture=sys option replaces sys.stdout/stderr with in-mem files. The --capture=fd option points file descriptors 1 and 2 to a temp file.

I'm including descriptions of sys and fd for completeness. But to be honest, I've never needed or used either. I frequently use -s. And to fully describe how -s works, I needed to touch on capture methods.

We don't have any print statements in our tests yet; a demo would be pointless. However, I encourage you to play with this a bit so you see it in action.

–lf, –last-failed

When one or more tests fails, having a convenient way to run just the failing tests is helpful for debugging. Just use --lf and you're ready to debug:

$ **cd /path/to/code/ch1**
$ **pytest --lf**
========== test session starts ==========
run-last-failure: rerun last 1 failures
collected 6 items

test_two.py F

========== FAILURES ==========
_____ test_failing _____

def test_failing():

```
> assert (1, 2, 3) == (3, 2, 1)
E assert (1, 2, 3) == (3, 2, 1)
E At index 0 diff: 1 != 3
E Use -v to get the full diff

test_two.py:2: AssertionError
==================== 5 tests deselected ====================
========= 1 failed, 5 deselected in 0.08 seconds ==========
```

This is great if you've been using a --tb option that hides some information and you want to re-run the failures with a different traceback option.

–ff, –failed-first

The --ff/--failed-first option will do the same as --last-failed, and then run the rest of the tests that passed last time:

```
$ cd /path/to/code/ch1
$ pytest --ff --tb=no
==================== test session starts ====================
run-last-failure: rerun last 1 failures first
collected 6 items

test_two.py F
test_one.py .
tasks/test_four.py ..
tasks/test_three.py ..

=========== 1 failed, 5 passed in 0.09 seconds ============
```

Usually, test_failing() from test_two.py is run after test_one.py. However, because test_failing() failed last time, --ff causes it to be run first.

-v, –verbose

The -v/--verbose option reports more information than without it. The most obvious difference is that each test gets its own line, and the name of the test and the outcome are spelled out instead of indicated with just a dot.

We've used it quite a bit already, but let's run it again for fun in conjunction with --ff and --tb=no:

```
$ cd /path/to/code/ch1
$ pytest -v --ff --tb=no
==================== test session starts ====================
```

```
run-last-failure: rerun last 1 failures first
collected 6 items

test_two.py::test_failing FAILED
test_one.py::test_passing PASSED
tasks/test_four.py::test_asdict PASSED
tasks/test_four.py::test_replace PASSED
tasks/test_three.py::test_defaults PASSED
tasks/test_three.py::test_member_access PASSED

========== 1 failed, 5 passed in 0.07 seconds ============
```

With color terminals, you'd see red FAILED and green PASSED outcomes in the report as well.

-q, –quiet

The -q/--quiet option is the opposite of -v/--verbose; it decreases the information reported. I like to use it in conjunction with --tb=line, which reports just the failing line of any failing tests.

Let's try -q by itself:

```
$ cd /path/to/code/ch1
$ pytest -q
.F....
======================== FAILURES ========================
_____________________ test_failing _____________________

def test_failing():
> assert (1, 2, 3) == (3, 2, 1)
E assert (1, 2, 3) == (3, 2, 1)
E At index 0 diff: 1 != 3
E Full diff:
E - (1, 2, 3)
E ? ^ ^
E + (3, 2, 1)
E ? ^ ^

test_two.py:2: AssertionError
1 failed, 5 passed in 0.08 seconds
```

The -q option makes the output pretty terse, but it's usually enough. We'll use the -q option frequently in the rest of the book (as well as --tb=no) to limit the output to what we are specifically trying to understand at the time.

-l, –showlocals

If you use the -l/--showlocals option, local variables and their values are displayed with tracebacks for failing tests.

So far, we don't have any failing tests that have local variables. If I take the test_replace() test and change

```
t_expected = Task('finish book', 'brian', True, 10)
```

to

```
t_expected = Task('finish book', 'brian', True, 11)
```

the 10 and 11 should cause a failure. Any change to the expected value will cause a failure. But this is enough to demonstrate the command-line option --l/--showlocals:

```
$ cd /path/to/code/ch1
$ pytest -l tasks
=================== test session starts ===================
collected 4 items

tasks/test_four.py .F
tasks/test_three.py ..




======================== FAILURES ========================
_____________________ test_replace _____________________

def test_replace():
t_before = Task('finish book', 'brian', False)
t_after = t_before._replace(id=10, done=True)
t_expected = Task('finish book', 'brian', True, 11)
> assert t_after == t_expected
E AssertionError: assert Task(summary=...e=True, id=10) == Task(
summary='...e=True, id=11)
E At index 3 diff: 10 != 11
E Use -v to get the full diff

t_after = Task(summary='finish book', owner='brian', done=True, id=10)
t_before = Task(summary='finish book', owner='brian', done=False, id=None)
t_expected = Task(summary='finish book', owner='brian', done=True, id=11)
```

```
tasks/test_four.py:20: AssertionError
========== 1 failed, 3 passed in 0.08 seconds ===========
```

The local variables t_after, t_before, and t_expected are shown after the code snippet, with the value they contained at the time of the failed assert.

–tb=style

The --tb=style option modifies the way tracebacks for failures are output. When a test fails, pytest lists the failures and what's called a traceback, which shows you the exact line where the failure occurred. Although tracebacks are helpful most of time, there may be times when they get annoying. That's where the --tb=style option comes in handy. The styles I find useful are short, line, and no. short prints just the assert line and the E evaluated line with no context; line keeps the failure to one line; no removes the traceback entirely.

Let's leave the modification to test_replace() to make it fail and run it with different traceback styles.

--tb=no removes the traceback entirely:

```
$ cd /path/to/code/ch1
$ pytest --tb=no tasks
================== test session starts ==================
collected 4 items

tasks/test_four.py .F
tasks/test_three.py ..

========== 1 failed, 3 passed in 0.04 seconds ===========
```

--tb=line in many cases is enough to tell what's wrong. If you have a ton of failing tests, this option can help to show a pattern in the failures:

```
$ pytest --tb=line tasks
================== test session starts ==================
collected 4 items

tasks/test_four.py .F
tasks/test_three.py ..

======================== FAILURES ========================
/path/to/code/ch1/tasks/test_four.py:20:
AssertionError: assert Task(summary=...e=True, id=10) == Task(
summary='...e=True, id=11)
========== 1 failed, 3 passed in 0.05 seconds ===========
```

The next step up in verbose tracebacks is --tb=short:

```
$ pytest --tb=short tasks
================== test session starts ==================
collected 4 items

tasks/test_four.py .F
tasks/test_three.py ..

======================= FAILURES ========================
_____________________ test_replace _____________________
tasks/test_four.py:20: in test_replace
assert t_after == t_expected
E AssertionError: assert Task(summary=...e=True, id=10) == Task(
summary='...e=True, id=11)
E At index 3 diff: 10 != 11
E Use -v to get the full diff
=========== 1 failed, 3 passed in 0.04 seconds ============
```

That's definitely enough to tell you what's going on.

There are three remaining traceback choices that we haven't covered so far.

pytest --tb=long will show you the most exhaustive, informative traceback possible. pytest --tb=auto will show you the long version for the first and last tracebacks, if you have multiple failures. This is the default behavior. pytest --tb=native will show you the standard library traceback without any extra information.

–durations=N

The --durations=N option is incredibly helpful when you're trying to speed up your test suite. It doesn't change how your tests are run; it reports the slowest N number of tests/setups/teardowns after the tests run. If you pass in --durations=0, it reports everything in order of slowest to fastest.

None of our tests are long, so I'll add a time.sleep(0.1) to one of the tests. Guess which one:

```
$ cd /path/to/code/ch1
$ pytest --durations=3 tasks
================= test session starts ==================
collected 4 items

tasks/test_four.py ..
tasks/test_three.py ..
```

```
============== slowest 3 test durations ===============
0.10s call tasks/test_four.py::test_replace
0.00s setup tasks/test_three.py::test_defaults
0.00s teardown tasks/test_three.py::test_member_access
============== 4 passed in 0.13 seconds ===============
```

The slow test with the extra sleep shows up right away with the label call, followed by setup and teardown. Every test essentially has three phases: call, setup, and teardown. Setup and teardown are also called fixtures and are a chance for you to add code to get data or the software system under test into a precondition state before the test runs, as well as clean up afterwards if necessary. I cover fixtures in depth in Chapter 3, *pytest Fixtures*.

–version

The --version option shows the version of pytest and the directory where it's installed:

```
$ pytest --version
This is pytest version 3.0.7, imported from
/path/to/venv/lib/python3.5/site-packages/pytest.py
```

Since we installed pytest into a virtual environment, pytest will be located in the site-packages directory of that virtual environment.

-h, –help

The -h/--help option is quite helpful, even after you get used to pytest. Not only does it show you how to use stock pytest, but it also expands as you install plugins to show options and configuration variables added by plugins.

The -h option shows:

- usage: pytest [options] [file_or_dir] [file_or_dir] [...]
- Command-line options and a short description, including options added via plugins
- A list of options available to ini style configuration files, which I'll discuss more in Chapter 6, *Configuration*
- A list of environmental variables that can affect pytest behavior (also discussed in Chapter 6, *Configuration*)
- A reminder that pytest --markers can be used to see available markers, discussed in Chapter 2, *Writing Test Functions*
- A reminder that pytest --fixtures can be used to see available fixtures, discussed in Chapter 3, *pytest Fixtures*

The last bit of information the help text displays is this note:

(shown according to specified file_or_dir or current dir if not specified)

This note is important because the options, markers, and fixtures can change based on which directory or test file you're running. This is because along the path to a specified file or directory, pytest may find conftest.py files that can include hook functions that create new options, fixture definitions, and marker definitions.

The ability to customize the behavior of pytest in conftest.py files and test files allows customized behavior local to a project or even a subset of the tests for a project. You'll learn about conftest.py and ini files such as pytest.ini in Chapter 6, *Configuration*.

Exercises

1. Create a new virtual environment using python -m virtualenv or python -m venv. Even if you know you don't need virtual environments for the project you're working on, humor me and learn enough about them to create one for trying out things in this book. I resisted using them for a very long time, and now I always use them. Read Appendix 1, *Virtual Environments* if you're having any difficulty.

2. Practice activating and deactivating your virtual environment a few times.

    - $ source venv/bin/activate
    - $ deactivate

    On Windows:

    - C:\Users\okken\sandbox>venv\scripts\activate.bat
    - C:\Users\okken\sandbox>deactivate

3. Install pytest in your new virtual environment. See Appendix 2, *pip* if you have any trouble. Even if you thought you already had pytest installed, you'll need to install it into the virtual environment you just created.

4. Create a few test files. You can use the ones we used in this chapter or make up your own. Practice running pytest against these files.

5. Change the assert statements. Don't just use assert something == something_else; try things like:

    - assert 1 in [2, 3, 4]
    - assert a < b
    - assert 'fizz' not in 'fizzbuzz'

What’s Next

In this chapter, we looked at where to get pytest and the various ways to run it. However, we didn’t discuss what goes into test functions. In the next chapter, we’ll look at writing test functions, parametrizing them so they get called with different data, and grouping tests into classes, modules, and packages.

Chapter 2
Writing Test Functions

In the last chapter, you got pytest up and running. You saw how to run it against files and directories and how many of the options worked. In this chapter, you'll learn how to write test functions in the context of testing a Python package. If you're using pytest to test something other than a Python package, most of this chapter still applies.

We're going to write tests for the Tasks package. Before we do that, I'll talk about the structure of a distributable Python package and the tests for it, and how to get the tests able to see the package under test. Then I'll show you how to use assert in tests, how tests handle unexpected exceptions, and testing for expected exceptions.

Eventually, we'll have a lot of tests. Therefore, you'll learn how to organize tests into classes, modules, and directories. I'll then show you how to use markers to mark which tests you want to run and discuss how builtin markers can help you skip tests and mark tests as expecting to fail. Finally, I'll cover parametrizing tests, which allows tests to get called with different data.

Testing a Package

We'll use the sample project, Tasks, as discussed in *The Tasks Project*, to see how to write test functions for a Python package. Tasks is a Python package that includes a command-line tool of the same name, tasks.

Appendix 4, *Packaging and Distributing Python Projects* includes an explanation of how to distribute your projects locally within a small team or globally through PyPI, so I won't go into detail of how to do that here; however, let's take a quick look at what's in the Tasks project and how the different files fit into the story of testing this project.

Following is the file structure for the Tasks project:

```
tasks_proj/
├── CHANGELOG.rst
├── LICENSE
├── MANIFEST.in
├── README.rst
├── setup.py
├── src
│   └── tasks
│       ├── __init__.py
│       ├── api.py
│       ├── cli.py
│       ├── config.py
│       ├── tasksdb_pymongo.py
│       └── tasksdb_tinydb.py
└── tests
    ├── conftest.py
    ├── pytest.ini
    ├── func
    │   ├── __init__.py
    │   ├── test_add.py
    │   └── ...
    └── unit
        ├── __init__.py
        ├── test_task.py
        └── ...
```

I included the complete listing of the project (with the exception of the full list of test files) to point out how the tests fit in with the rest of the project, and to point out a few files that are of key importance to testing, namely conftest.py, pytest.ini, the various __init__.py files, and setup.py.

All of the tests are kept in tests and separate from the package source files in src. This isn't a requirement of pytest, but it's a best practice.

All of the top-level files, CHANGELOG.rst, LICENSE, README.rst, MANIFEST.in, and setup.py, are discussed in more detail in Appendix 4, *Packaging and Distributing Python Projects*. Although setup.py is important for building a distribution out of a package, it's also crucial for being able to install a package locally so that the package is available for import.

Functional and unit tests are separated into their own directories. This is an arbitrary decision and not required. However, organizing test files into multiple directories allows you to easily run a subset of tests. I like to keep functional and unit tests separate because functional tests should only break if we are intentionally changing functionality of the system, whereas unit tests could break during a refactoring or an implementation change.

The project contains two types of __init__.py files: those found under the src/ directory and those found under tests/. The src/tasks/__init__.py file tells Python that the directory is a package. It also acts as the main interface to the package when someone uses import tasks. It contains code to import specific functions from api.py so that cli.py and our test files can access package functionality like tasks.add() instead of having to do tasks.api.add().

The tests/func/__init__.py and tests/unit/__init__.py files are empty. They tell pytest to go up one directory to look for the root of the test directory and to look for the pytest.ini file.

The pytest.ini file is optional. It contains project-wide pytest configuration. There should be at most only one of these in your project. It can contain directives that change the behavior of pytest, such as setting up a list of options that will always be used. You'll learn all about pytest.ini in Chapter 6, *Configuration*.

The conftest.py file is also optional. It is considered by pytest as a "local plugin" and can contain hook functions and fixtures. Hook functions are a way to insert code into part of the pytest execution process to alter how pytest works. Fixtures are setup and teardown functions that run before and after test functions, and can be used to represent resources and data used by the tests. (Fixtures are discussed in Chapter 3, *pytest Fixtures* and Chapter 4, *Builtin Fixtures*, and hook functions are discussed in Chapter 5, *Plugins*.) Hook functions and fixtures that are used by tests in multiple subdirectories should be contained in tests/conftest.py. You can have multiple conftest.py files; for example, you can have one at tests and one for each subdirectory under tests.

If you haven't already done so, you can download a copy of the source code for this project on the book's website.[5] Alternatively, you can work on your own project with a similar structure.

Installing a Package Locally

The test file, tests/test_task.py, contains the tests we worked on in *Running pytest*, in files test_three.py and test_four.py. I've just renamed it here to something that makes more sense for what it's testing and copied everything into one file. I also removed the definition of the Task data structure, because that really belongs in api.py.

Here is test_task.py:

ch2/tasks_proj/tests/unit/test_task.py

```
    """Test the Task data type."""
    from tasks import Task


    def test_asdict():
    """_asdict() should return a dictionary."""
    t_task = Task('do something', 'okken', True, 21)
    t_dict = t_task._asdict()
    expected = {'summary': 'do something',
    'owner': 'okken',
    'done': True,
    'id': 21}
    assert t_dict == expected


    def test_replace():
    """replace() should change passed in fields."""
    t_before = Task('finish book', 'brian', False)
    t_after = t_before._replace(id=10, done=True)
    t_expected = Task('finish book', 'brian', True, 10)
    assert t_after == t_expected


    def test_defaults():
    """Using no parameters should invoke defaults."""
    t1 = Task()
    t2 = Task(None, None, False, None)
    assert t1 == t2


    def test_member_access():
    """Check .field functionality of namedtuple."""
    t = Task('buy milk', 'brian')
    assert t.summary == 'buy milk'
    assert t.owner == 'brian'
    assert (t.done, t.id) == (False, None)
```

The test_task.py file has this import statement:

```
    from tasks import Task
```

The best way to allow the tests to be able to import tasks or from tasks import something is to install tasks locally using pip. This is possible because there’s a setup.py file present to direct

pip.

Install tasks either by running pip install . or pip install -e . from the tasks_proj directory. Or you can run pip install -e tasks_proj from one directory up:

```
$ cd /path/to/code
$ pip install ./tasks_proj/
$ pip install --no-cache-dir ./tasks_proj/
Processing ./tasks_proj
Collecting click (from tasks==0.1.0)
Downloading click-6.7-py2.py3-none-any.whl (71kB)
 ...
Collecting tinydb (from tasks==0.1.0)
Downloading tinydb-3.4.0.tar.gz
Collecting six (from tasks==0.1.0)
Downloading six-1.10.0-py2.py3-none-any.whl
Installing collected packages: click, tinydb, six, tasks
Running setup.py install for tinydb ... done
Running setup.py install for tasks ... done
Successfully installed click-6.7 six-1.10.0 tasks-0.1.0 tinydb-3.4.0
```

If you only want to run tests against tasks, this command is fine. If you want to be able to modify the source code while tasks is installed, you need to install it with the -e option (for “editable”):

```
$ pip install -e ./tasks_proj/
Obtaining file:///path/to/code/tasks_proj
Requirement already satisfied: click in
/path/to/venv/lib/python3.6/site-packages (from tasks==0.1.0)
Requirement already satisfied: tinydb in
/path/to/venv/lib/python3.6/site-packages (from tasks==0.1.0)
Requirement already satisfied: six in
/path/to/venv/lib/python3.6/site-packages (from tasks==0.1.0)
Installing collected packages: tasks
Found existing installation: tasks 0.1.0
Uninstalling tasks-0.1.0:
Successfully uninstalled tasks-0.1.0
Running setup.py develop for tasks
Successfully installed tasks
```

Now let’s try running tests:

```
$ cd /path/to/code/ch2/tasks_proj/tests/unit
$ pytest test_task.py
==================== test session starts ====================
```

```
collected 4 items

test_task.py ....

==================== 4 passed in 0.01 seconds ====================
```

The import worked! The rest of our tests can now safely use import tasks. Now let’s write some tests.

Using assert Statements

When you write test functions, the normal Python assert statement is your primary tool to communicate test failure. The simplicity of this within pytest is brilliant. It's what drives a lot of developers to use pytest over other frameworks.

If you've used any other testing framework, you've probably seen various assert helper functions. For example, the following is a list of a few of the assert forms and assert helper functions:

pytest	unittest
assert something	assertTrue(something)
assert a == b	assertEqual(a, b)
assert a <= b	assertLessEqual(a, b)
…	…

With pytest, you can use assert <expression> with any expression. If the expression would evaluate to False if converted to a bool, the test would fail.

pytest includes a feature called assert rewriting that intercepts assert calls and replaces them with something that can tell you more about why your assertions failed. Let's see how helpful this rewriting is by looking at a few assertion failures:

ch2/tasks_proj/tests/unit/test_task_fail.py

```
"""Use the Task type to show test failures."""
from tasks import Task


def test_task_equality():
"""Different tasks should not be equal."""
t1 = Task('sit there', 'brian')
t2 = Task('do something', 'okken')
assert t1 == t2


def test_dict_equality():
```

```
    """Different tasks compared as dicts should not be equal."""
    t1_dict = Task('make sandwich', 'okken')._asdict()
    t2_dict = Task('make sandwich', 'okkem')._asdict()
    assert t1_dict == t2_dict
```

All of these tests fail, but what's interesting is the traceback information:

```
$ cd /path/to/code/ch2/tasks_proj/tests/unit
$ pytest test_task_fail.py
==================== test session starts ====================
collected 2 items

test_task_fail.py FF

========================== FAILURES ==========================
_____________________ test_task_equality _____________________

def test_task_equality():
t1 = Task('sit there', 'brian')
t2 = Task('do something', 'okken')
> assert t1 == t2
E AssertionError: assert Task(summary=...alse, id=None) ==
Task(summary='...alse, id=None)
E At index 0 diff: 'sit there' != 'do something'
E Use -v to get the full diff

test_task_fail.py:6: AssertionError
_____________________ test_dict_equality _____________________

def test_dict_equality():
t1_dict = Task('make sandwich', 'okken')._asdict()
t2_dict = Task('make sandwich', 'okkem')._asdict()
> assert t1_dict == t2_dict
E AssertionError: assert OrderedDict([...('id', None)]) ==
OrderedDict([(...('id', None)])
E Omitting 3 identical items, use -v to show
E Differing items:
E {'owner': 'okken'} != {'owner': 'okkem'}
E Use -v to get the full diff

test_task_fail.py:11: AssertionError
================== 2 failed in 0.06 seconds ==================
```

Wow. That's a lot of information. For each failing test, the exact line of failure is shown with a > pointing to the failure. The E lines show you extra information about the assert failure to help you figure out what went wrong.

I intentionally put two mismatches in test_task_equality(), but only the first was shown in the previous code. Let's try it again with the -v flag, as suggested in the error message:

```
$ pytest -v test_task_fail.py::test_task_equality
==================== test session starts ====================
collected 3 items

test_task_fail.py::test_task_equality FAILED

========================== FAILURES ==========================
_____________________ test_task_equality _____________________

def test_task_equality():
t1 = Task('sit there', 'brian')
t2 = Task('do something', 'okken')
> assert t1 == t2
E AssertionError: assert Task(summary=...alse, id=None) ==
Task(summary='...alse, id=None)
E At index 0 diff: 'sit there' != 'do something'
E Full diff:
E - Task(summary='sit there', owner='brian', done=False, id=None)
E ? ^^^ ^^^ ^^^^
E + Task(summary='do something', owner='okken', done=False, id=None)
E ? +++ ^^^ ^^^ ^^^^

test_task_fail.py:6: AssertionError
================== 1 failed in 0.07 seconds ==================
```

Well, I think that's pretty darned cool. pytest not only found both differences, but it also showed us exactly where the differences are.

This example only used equality assert; many more varieties of assert statements with awesome trace debug information are found on the pytest.org website.[6]

Expecting Exceptions

Exceptions may be raised in a few places in the Tasks API. Let's take a quick peek at the functions found in tasks/api.py:

```
def add(task): # type: (Task) -> int
def get(task_id): # type: (int) -> Task
def list_tasks(owner=None): # type: (str|None) -> list of Task
def count(): # type: (None) -> int
def update(task_id, task): # type: (int, Task) -> None
def delete(task_id): # type: (int) -> None
def delete_all(): # type: () -> None
def unique_id(): # type: () -> int
def start_tasks_db(db_path, db_type): # type: (str, str) -> None
def stop_tasks_db(): # type: () -> None
```

There's an agreement between the CLI code in cli.py and the API code in api.py as to what types will be sent to the API functions. These API calls are a place where I'd expect exceptions to be raised if the type is wrong.

To make sure these functions raise exceptions if called incorrectly, let's use the wrong type in a test function to intentionally cause TypeError exceptions, and use with pytest.raises(<expected exception>), like this:

ch2/tasks_proj/tests/func/test_api_exceptions.py

```
import pytest
import tasks


def test_add_raises():
"""add() should raise an exception with wrong type param."""
with pytest.raises(TypeError):
tasks.add(task='not a Task object')
```

In test_add_raises(), the with pytest.raises(TypeError): statement says that whatever is in the next block of code should raise a TypeError exception. If no exception is raised, the test fails. If the test raises a different exception, it fails.

We just checked for the type of exception in test_add_raises(). You can also check the parameters to the exception. For start_tasks_db(db_path, db_type), not only does db_type need to be a string, it really has to be either 'tiny' or 'mongo'. You can check to make sure the exception message is correct by adding as excinfo:

ch2/tasks_proj/tests/func/test_api_exceptions.py

```
def test_start_tasks_db_raises():
```

```
    """Make sure unsupported db raises an exception."""
    with pytest.raises(ValueError) as excinfo:
    tasks.start_tasks_db('some/great/path', 'mysql')
    exception_msg = excinfo.value.args[0]
    assert exception_msg == "db_type must be a 'tiny' or 'mongo'"
```

This allows us to look at the exception more closely. The variable name you put after as (excinfo in this case) is filled with information about the exception, and is of type ExceptionInfo.

In our case, we want to make sure the first (and only) parameter to the exception matches a string.

Marking Test Functions

pytest provides a cool mechanism to let you put markers on test functions. A test can have more than one marker, and a marker can be on multiple tests.

Markers make sense after you see them in action. Let's say we want to run a subset of our tests as a quick "smoke test" to get a sense for whether or not there is some major break in the system. Smoke tests are by convention not all-inclusive, thorough test suites, but a select subset that can be run quickly and give a developer a decent idea of the health of all parts of the system.

To add a smoke test suite to the Tasks project, we can add @mark.pytest.smoke to some of the tests. Let's add it to a couple of tests in test_api_exceptions.py (note that the markers smoke and get aren't built into pytest; I just made them up):

ch2/tasks_proj/tests/func/test_api_exceptions.py

```
@pytest.mark.smoke
def test_list_raises():
"""list() should raise an exception with wrong type param."""
with pytest.raises(TypeError):
tasks.list_tasks(owner=123)


@pytest.mark.get
@pytest.mark.smoke
def test_get_raises():
"""get() should raise an exception with wrong type param."""
with pytest.raises(TypeError):
tasks.get(task_id='123')
```

Now, let's run just those tests that are marked with -m marker_name:

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest -v -m 'smoke' test_api_exceptions.py
==================== test session starts ====================
collected 7 items

test_api_exceptions.py::test_list_raises PASSED
test_api_exceptions.py::test_get_raises PASSED

====================== 5 tests deselected ======================
============ 2 passed, 5 deselected in 0.03 seconds ============
$ pytest -v -m 'get' test_api_exceptions.py
==================== test session starts ====================
```

```
collected 7 items

test_api_exceptions.py::test_get_raises PASSED

====================== 6 tests deselected ======================
============ 1 passed, 6 deselected in 0.01 seconds ============
```

Remember that -v is short for --verbose and lets us see the names of the tests that are run. Using -m 'smoke' runs both tests marked with @pytest.mark.smoke. Using -m 'get' runs the one test marked with @pytest.mark.get. Pretty straightforward.

It gets better. The expression after -m can use and, or, and not to combine multiple markers:

```
$ pytest -v -m 'smoke and get' test_api_exceptions.py
==================== test session starts ====================
collected 7 items

test_api_exceptions.py::test_get_raises PASSED

====================== 6 tests deselected ======================
============ 1 passed, 6 deselected in 0.03 seconds ============
```

That time we only ran the test that had both smoke and get markers. We can use not as well:

```
$ pytest -v -m 'smoke and not get' test_api_exceptions.py
==================== test session starts ====================
collected 7 items

test_api_exceptions.py::test_list_raises PASSED

====================== 6 tests deselected ======================
============ 1 passed, 6 deselected in 0.03 seconds ============
```

The addition of -m 'smoke and not get' selected the test that was marked with @pytest.mark.smoke but not @pytest.mark.get.

Filling Out the Smoke Test

The previous tests don't seem like a reasonable smoke test suite yet. We haven't actually touched the database or added any tasks. Surely a smoke test would do that.

Let's add a couple of tests that look at adding a task, and use one of them as part of our smoke test suite:

ch2/tasks_proj/tests/func/test_add.py

```
import pytest
```

```
import tasks
from tasks import Task


def test_add_returns_valid_id():
"""tasks.add(<valid task>) should return an integer."""
# GIVEN an initialized tasks db
# WHEN a new task is added
# THEN returned task_id is of type int
new_task = Task('do something')
task_id = tasks.add(new_task)
assert isinstance(task_id, int)


@pytest.mark.smoke
def test_added_task_has_id_set():
"""Make sure the task_id field is set by tasks.add()."""
# GIVEN an initialized tasks db
# AND a new task is added
new_task = Task('sit in chair', owner='me', done=True)
task_id = tasks.add(new_task)

# WHEN task is retrieved
task_from_db = tasks.get(task_id)

# THEN task_id matches id field
assert task_from_db.id == task_id
```

Both of these tests have the comment GIVEN an initialized tasks db, and yet there is no database initialized in the test. We can define a fixture to get the database initialized before the test and cleaned up after the test:

ch2/tasks_proj/tests/func/test_add.py

```
@pytest.fixture(autouse=True)
def initialized_tasks_db(tmpdir):
"""Connect to db before testing, disconnect after."""
# Setup : start db
tasks.start_tasks_db(str(tmpdir), 'tiny')

yield # this is where the testing happens

# Teardown : stop db
tasks.stop_tasks_db()
```

The fixture, tmpdir, used in this example is a builtin fixture. You'll learn all about builtin fixtures in Chapter 4, *Builtin Fixtures*, and you'll learn about writing your own fixtures and how they work in Chapter 3, *pytest Fixtures*, including the autouse parameter used here.

autouse as used in our test indicates that all tests in this file will use the fixture. The code before the yield runs before each test; the code after the yield runs after the test. The yield can return data to the test if desired. You'll look at all that and more in later chapters, but here we need some way to set up the database for testing, so I couldn't wait any longer to show you a fixture. (pytest also supports old-fashioned setup and teardown functions, like what is used in unittest and nose, but they are not nearly as fun. However, if you are curious, they are described in Appendix 5, *xUnit Fixtures*.)

Let's set aside fixture discussion for now and go to the top of the project and run our smoke test suite:

```
$ cd /path/to/code/ch2/tasks_proj
$ pytest -v -m 'smoke'
====================== test session starts ======================
collected 56 items

tests/func/test_add.py::test_added_task_has_id_set PASSED
tests/func/test_api_exceptions.py::test_list_raises PASSED
tests/func/test_api_exceptions.py::test_get_raises PASSED

====================== 53 tests deselected ======================
=========== 3 passed, 53 deselected in 0.11 seconds ============
```

This shows that marked tests from different files can all run together.

Skipping Tests

While the markers discussed in *Marking Test Functions* were names of your own choosing, pytest includes a few helpful builtin markers: skip, skipif, and xfail. I'll discuss skip and skipif in this section, and xfail in the next.

The skip and skipif markers enable you to skip tests you don't want to run. For example, let's say we weren't sure how tasks.unique_id() was supposed to work. Does each call to it return a different number? Or is it just a number that doesn't exist in the database already?

First, let's write a test (note that the initialized_tasks_db fixture is in this file, too; it's just not shown here):

ch2/tasks_proj/tests/func/test_unique_id_1.py

```
import pytest
import tasks


def test_unique_id():
"""Calling unique_id() twice should return different numbers."""
id_1 = tasks.unique_id()
id_2 = tasks.unique_id()
assert id_1 != id_2
```

Then give it a run:

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest test_unique_id_1.py
===================== test session starts ======================
collected 1 item s

test_unique_id_1.py F

============================ FAILURES ============================
_____________________________ test_unique_id _____________________________

def test_unique_id():
"""Calling unique_id() twice should return different numbers."""
id_1 = tasks.unique_id()
id_2 = tasks.unique_id()
> assert id_1 != id_2
E assert 1 != 1
```

```
test_unique_id_1.py:12: AssertionError
================== 1 failed in 0.06 seconds ==================
```

Hmm. Maybe we got that wrong. After looking at the API a bit more, we see that the docstring says """Return an integer that does not exist in the db.""".

We could just change the test. But instead, let's just mark the first one to get skipped for now:

ch2/tasks_proj/tests/func/test_unique_id_2.py

```
@pytest.mark.skip(reason='misunderstood the API')
def test_unique_id_1():
    """Calling unique_id() twice should return different numbers."""
    id_1 = tasks.unique_id()
    id_2 = tasks.unique_id()
    assert id_1 != id_2


def test_unique_id_2():
    """unique_id() should return an unused id."""
    ids = []
    ids.append(tasks.add(Task('one')))
    ids.append(tasks.add(Task('two')))
    ids.append(tasks.add(Task('three')))
    # grab a unique id
    uid = tasks.unique_id()
    # make sure it isn't in the list of existing ids
    assert uid not in ids
```

Marking a test to be skipped is as simple as adding @pytest.mark.skip() just above the test function.

Let's run again:

```
$ pytest -v test_unique_id_2.py
============================ test session starts ============================
collected 2 items

test_unique_id_2.py::test_unique_id_1 SKIPPED
test_unique_id_2.py::test_unique_id_2 PASSED

================== 1 passed, 1 skipped in 0.02 seconds ==================
```

Now, let's say that for some reason we decide the first test should be valid also, and we intend to make that work in version 0.2.0 of the package. We can leave the test in place and use skipif instead:

ch2/tasks_proj/tests/func/test_unique_id_3.py

```
@pytest.mark.skipif(tasks.__version__ < '0.2.0',
reason='not supported until version 0.2.0')
def test_unique_id_1():
"""Calling unique_id() twice should return different numbers."""
id_1 = tasks.unique_id()
id_2 = tasks.unique_id()
assert id_1 != id_2
```

The expression we pass into skipif() can be any valid Python expression. In this case, we're checking the package version.

We included reasons in both skip and skipif. It's not required in skip, but it is required in skipif. I like to include a reason for every skip, skipif, or xfail.

Here's the output of the changed code:

```
$ pytest test_unique_id_3.py
========================== test session starts ==========================
collected 2 items

test_unique_id_3.py s.

=================== 1 passed, 1 skipped in 0.02 seconds ===================
```

The s. shows that one test was skipped and one test passed.

We can see which one with -v:

```
$ pytest -v test_unique_id_3.py
========================== test session starts ==========================
collected 2 items

test_unique_id_3.py::test_unique_id_1 SKIPPED
test_unique_id_3.py::test_unique_id_2 PASSED

=================== 1 passed, 1 skipped in 0.03 seconds ===================
```

But we still don't know why. We can see those reasons with -rs:

```
$ pytest -rs test_unique_id_3.py
======================== test session starts ========================
collected 2 items

test_unique_id_3.py s.
```

```
========================= short test summary info ==========================
SKIP [1] func/test_unique_id_3.py:5: not supported until version 0.2.0

================== 1 passed, 1 skipped in 0.03 seconds ==================
```

The -r chars option has this help text:

```
$ pytest --help
...
-r chars

show extra test summary info as specified by chars
(f)ailed, (E)error, (s)skipped, (x)failed, (X)passed,
(p)passed, (P)passed with output, (a)all except pP.
...
```

It’s not only helpful for understanding test skips, but also you can use it for other test outcomes as well.

Marking Tests as Expecting to Fail

With the skip and skipif markers, a test isn't even attempted if skipped. With the xfail marker, we are telling pytest to run a test function, but that we expect it to fail. Let's modify our unique_id() test again to use xfail:

ch2/tasks_proj/tests/func/test_unique_id_4.py

```
@pytest.mark.xfail(tasks.__version__ < '0.2.0',
reason='not supported until version 0.2.0')
def test_unique_id_1():
"""Calling unique_id() twice should return different numbers."""
id_1 = tasks.unique_id()
id_2 = tasks.unique_id()
assert id_1 != id_2


@pytest.mark.xfail()
def test_unique_id_is_a_duck():
"""Demonstrate xfail."""
uid = tasks.unique_id()
assert uid == 'a duck'


@pytest.mark.xfail()
def test_unique_id_not_a_duck():
"""Demonstrate xpass."""
uid = tasks.unique_id()
assert uid != 'a duck'
```

The first test is the same as before, but with xfail. The next two tests are listed as xfail, and differ only by == vs. !=. So one of them is bound to pass.

Running this shows:

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest test_unique_id_4.py
======================== test session starts ========================
collected 4 items

test_unique_id_4.py xxX.

========== 1 passed, 2 xfailed, 1 xpassed in 0.07 seconds ===========
```

The x is for XFAIL, which means “expected to fail.” The capital X is for XPASS or “expected to fail but passed.”

--verbose lists longer descriptions:

```
$ pytest -v test_unique_id_4.py
======================== test session starts ========================
collected 4 items

test_unique_id_4.py::test_unique_id_1 xfail
test_unique_id_4.py::test_unique_id_is_a_duck xfail
test_unique_id_4.py::test_unique_id_not_a_duck XPASS
test_unique_id_4.py::test_unique_id_2 PASSED

========== 1 passed, 2 xfailed, 1 xpassed in 0.08 seconds ===========
```

You can configure pytest to report the tests that pass but were marked with xfail to be reported as FAIL. This is done in a pytest.ini file:

```
[pytest]
xfail_strict=true
```

I’ll discuss pytest.ini more in Chapter 6, *Configuration*.

Running a Subset of Tests

I've talked about how you can place markers on tests and run tests based on markers. You can run a subset of tests in several other ways. You can run all of the tests, or you can select a single directory, file, class within a file, or an individual test in a file or class. You haven't seen test classes used yet, so you'll look at one in this section. You can also use an expression to match test names. Let's take a look at these.

A Single Directory

To run all the tests from one directory, use the directory as a parameter to pytest:

```
$ cd /path/to/code/ch2/tasks_proj
$ pytest tests/func --tb=no
====================== test session starts ======================
collected 50 items

tests/func/test_add.py ..
tests/func/test_add_variety.py .............................
tests/func/test_api_exceptions.py .......
tests/func/test_unique_id_1.py F
tests/func/test_unique_id_2.py s.
tests/func/test_unique_id_3.py s.
tests/func/test_unique_id_4.py xxX.

1 failed, 44 passed, 2 skipped, 2 xfailed, 1 xpassed in 0.26 seconds
```

An important trick to learn is that using -v gives you the syntax for how to run a specific directory, class, and test.

```
$ pytest -v tests/func --tb=no
====================== test session starts ======================
collected 50 items

tests/func/test_add.py::test_add_returns_valid_id PASSED
tests/func/test_add.py::test_added_task_has_id_set PASSED
...
tests/func/test_api_exceptions.py::test_add_raises PASSED
tests/func/test_api_exceptions.py::test_list_raises PASSED
tests/func/test_api_exceptions.py::test_get_raises PASSED
...
tests/func/test_unique_id_1.py::test_unique_id FAILED
```

```
tests/func/test_unique_id_2.py::test_unique_id_1 SKIPPED
tests/func/test_unique_id_2.py::test_unique_id_2 PASSED
...
tests/func/test_unique_id_4.py::test_unique_id_1 xfail
tests/func/test_unique_id_4.py::test_unique_id_is_a_duck xfail
tests/func/test_unique_id_4.py::test_unique_id_not_a_duck XPASS
tests/func/test_unique_id_4.py::test_unique_id_2 PASSED

1 failed, 44 passed, 2 skipped, 2 xfailed, 1 xpassed in 0.30 seconds
```

You'll see the syntax listed here in the next few examples.

A Single Test File/Module

To run a file full of tests, list the file with the relative path as a parameter to pytest:

```
$ cd /path/to/code/ch2/tasks_proj
$ pytest tests/func/test_add.py
============================= test session starts ==============================
collected 2 items

tests/func/test_add.py ..

=========================== 2 passed in 0.05 seconds ===========================
```

We've been doing this for a while.

A Single Test Function

To run a single test function, add :: and the test function name:

```
$ cd /path/to/code/ch2/tasks_proj
$ pytest -v tests/func/test_add.py::test_add_returns_valid_id
============================= test session starts ==============================
collected 3 items

tests/func/test_add.py::test_add_returns_valid_id PASSED

=========================== 1 passed in 0.02 seconds ===========================
```

Use -v so you can see which function was run.

A Single Test Class

Test classes are a way to group tests that make sense to be grouped together. Here's an example:

ch2/tasks_proj/tests/func/test_api_exceptions.py

```
class TestUpdate():
"""Test expected exceptions with tasks.update()."""

def test_bad_id(self):
"""A non-int id should raise an excption."""
with pytest.raises(TypeError):
tasks.update(task_id={'dict instead': 1},
task=tasks.Task())

def test_bad_task(self):
"""A non-Task task should raise an excption."""
with pytest.raises(TypeError):
tasks.update(task_id=1, task='not a task')
```

Since these are two related tests that both test the update() function, it's reasonable to group them in a class. To run just this class, do like we did with functions and add ::, then the class name to the file parameter:

```
$ cd /path/to/code/ch2/tasks_proj
$ pytest -v tests/func/test_api_exceptions.py::TestUpdate
========================== test session starts ==========================
collected 7 items

tests/func/test_api_exceptions.py::TestUpdate::test_bad_id PASSED
tests/func/test_api_exceptions.py::TestUpdate::test_bad_task PASSED

======================== 2 passed in 0.03 seconds ========================
```

A Single Test Method of a Test Class

If you don't want to run all of a test class—just one method—just add another :: and the method name:

```
$ cd /path/to/code/ch2/tasks_proj
$ pytest -v tests/func/test_api_exceptions.py::TestUpdate::test_bad_id
====================== test session starts ======================
collected 1 item

tests/func/test_api_exceptions.py::TestUpdate::test_bad_id PASSED

================== 1 passed in 0.03 seconds ==================
```

Grouping Syntax Shown by Verbose Listing

Remember that the syntax for how to run a subset of tests by directory, file, function, class, and method doesn't have to be memorized. The format is the same as the test function listing when you run pytest -v.

A Set of Tests Based on Test Name

The -k option enables you to pass in an expression to run tests that have certain names specified by the expression as a substring of the test name. You can use and, or, and not in your expression to create complex expressions.

For example, we can run all of the functions that have _raises in their name:

```
$ cd /path/to/code/ch2/tasks_proj
$ pytest -v -k _raises
==================== test session starts ====================
collected 56 items

tests/func/test_api_exceptions.py::test_add_raises PASSED
tests/func/test_api_exceptions.py::test_list_raises PASSED
tests/func/test_api_exceptions.py::test_get_raises PASSED
tests/func/test_api_exceptions.py::test_delete_raises PASSED
tests/func/test_api_exceptions.py::test_start_tasks_db_raises PASSED

==================== 51 tests deselected ====================
========== 5 passed, 51 deselected in 0.07 seconds ===========
```

We can use and and not to get rid of the test_delete_raises() from the session:

```
$ pytest -v -k "_raises and not delete"
==================== test session starts ====================
collected 56 items

tests/func/test_api_exceptions.py::test_add_raises PASSED
tests/func/test_api_exceptions.py::test_list_raises PASSED
tests/func/test_api_exceptions.py::test_get_raises PASSED
tests/func/test_api_exceptions.py::test_start_tasks_db_raises PASSED

==================== 52 tests deselected ====================
========== 4 passed, 52 deselected in 0.06 seconds ===========
```

In this section, you learned how to run specific test files, directories, classes, and functions, and how to use expressions with -k to run specific sets of tests. In the next section, you'll learn how one test function can turn into many test cases by allowing the test to run multiple times with different test data.

Parametrized Testing

Sending some values through a function and checking the output to make sure it's correct is a common pattern in software testing. However, calling a function once with one set of values and one check for correctness isn't enough to fully test most functions. Parametrized testing is a way to send multiple sets of data through the same test and have pytest report if any of the sets failed.

To help understand the problem parametrized testing is trying to solve, let's take a simple test for add():

ch2/tasks_proj/tests/func/test_add_variety.py

```
import pytest
import tasks
from tasks import Task


def test_add_1():
"""tasks.get() using id returned from add() works."""
task = Task('breathe', 'BRIAN', True)
task_id = tasks.add(task)
t_from_db = tasks.get(task_id)
# everything but the id should be the same
assert equivalent(t_from_db, task)


def equivalent(t1, t2):
"""Check two tasks for equivalence."""
# Compare everything but the id field
return ((t1.summary == t2.summary) and
(t1.owner == t2.owner) and
(t1.done == t2.done))


@pytest.fixture(autouse=True)
def initialized_tasks_db(tmpdir):
"""Connect to db before testing, disconnect after."""
tasks.start_tasks_db(str(tmpdir), 'tiny')
yield
tasks.stop_tasks_db()
```

When a Task object is created, its id field is set to None. After it's added and retrieved from the database, the id field will be set. Therefore, we can't just use == to check to see if our task was added and retrieved correctly. The equivalent() helper function checks all but the id field. The

autouse fixture is included to make sure the database is accessible. Let's make sure the test passes:

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest -v test_add_variety.py::test_add_1
==================== test session starts ====================
collected 1 item

test_add_variety.py::test_add_1 PASSED

=================== 1 passed in 0.03 seconds ===================
```

The test seems reasonable. However, it's just testing one example task. What if we want to test lots of variations of a task? No problem. We can use @pytest.mark.parametrize(argnames, argvalues) to pass lots of data through the same test, like this:

ch2/tasks_proj/tests/func/test_add_variety.py

```
@pytest.mark.parametrize('task',
[Task('sleep', done=True),
Task('wake', 'brian'),
Task('breathe', 'BRIAN', True),
Task('exercise', 'BrIaN', False)])
def test_add_2(task):
"""Demonstrate parametrize with one parameter."""
task_id = tasks.add(task)
t_from_db = tasks.get(task_id)
assert equivalent(t_from_db, task)
```

The first argument to parametrize() is a string with a comma-separated list of names—'task', in our case. The second argument is a list of values, which in our case is a list of Task objects. pytest will run this test once for each task and report each as a separate test:

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest -v test_add_variety.py::test_add_2
==================== test session starts ====================
collected 4 items

test_add_variety.py::test_add_2[task0] PASSED
test_add_variety.py::test_add_2[task1] PASSED
test_add_variety.py::test_add_2[task2] PASSED
test_add_variety.py::test_add_2[task3] PASSED

=================== 4 passed in 0.05 seconds ===================
```

This use of parametrize() works for our purposes. However, let's pass in the tasks as tuples to see

how multiple test parameters would work:

ch2/tasks_proj/tests/func/test_add_variety.py

```
@pytest.mark.parametrize('summary, owner, done',
[('sleep', None, False),
('wake', 'brian', False),
('breathe', 'BRIAN', True),
('eat eggs', 'BrIaN', False),
])
def test_add_3(summary, owner, done):
"""Demonstrate parametrize with multiple parameters."""
task = Task(summary, owner, done)
task_id = tasks.add(task)
t_from_db = tasks.get(task_id)
assert equivalent(t_from_db, task)
```

When you use types that are easy for pytest to convert into strings, the test identifier uses the parameter values in the report to make it readable:

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest -v test_add_variety.py::test_add_3
==================== test session starts ====================
collected 4 items

test_add_variety.py::test_add_3[sleep-None-False] PASSED
test_add_variety.py::test_add_3[wake-brian-False] PASSED
test_add_variety.py::test_add_3[breathe-BRIAN-True] PASSED
test_add_variety.py::test_add_3[eat eggs-BrIaN-False] PASSED

=================== 4 passed in 0.05 seconds ===================
```

You can use that whole test identifier—called a node in pytest terminology—to re-run the test if you want:

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest -v test_add_variety.py::test_add_3[sleep-None-False]
==================== test session starts ====================
collected 1 item

test_add_variety.py::test_add_3[sleep-None-False] PASSED

=================== 1 passed in 0.02 seconds ===================
```

Be sure to use quotes if there are spaces in the identifier:

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest -v "test_add_variety.py::test_add_3[eat eggs-BrIaN-False]"
==================== test session starts ====================
collected 1 item

test_add_variety.py::test_add_3[eat eggs-BrIaN-False] PASSED

=================== 1 passed in 0.03 seconds ===================
```

Now let's go back to the list of tasks version, but move the task list to a variable outside the function:

ch2/tasks_proj/tests/func/test_add_variety.py

```
tasks_to_try = (Task('sleep', done=True),
Task('wake', 'brian'),
Task('wake', 'brian'),
Task('breathe', 'BRIAN', True),
Task('exercise', 'BrIaN', False))


@pytest.mark.parametrize('task', tasks_to_try)
def test_add_4(task):
"""Slightly different take."""
task_id = tasks.add(task)
t_from_db = tasks.get(task_id)
assert equivalent(t_from_db, task)
```

It's convenient and the code looks nice. But the readability of the output is hard to interpret:

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest -v test_add_variety.py::test_add_4
==================== test session starts ====================
collected 5 items

test_add_variety.py::test_add_4[task0] PASSED
test_add_variety.py::test_add_4[task1] PASSED
test_add_variety.py::test_add_4[task2] PASSED
test_add_variety.py::test_add_4[task3] PASSED
test_add_variety.py::test_add_4[task4] PASSED

=================== 5 passed in 0.05 seconds ===================
```

The readability of the multiple parameter version is nice, but so is the list of Task objects. To compromise, we can use the ids optional parameter to parametrize() to make our own identifiers

for each task data set. The ids parameter needs to be a list of strings the same length as the number of data sets. However, because we assigned our data set to a variable name, tasks_to_try, we can use it to generate ids:

ch2/tasks_proj/tests/func/test_add_variety.py

```
task_ids = ['Task({},{},{})'.format(t.summary, t.owner, t.done)
for t in tasks_to_try]


@pytest.mark.parametrize('task', tasks_to_try, ids=task_ids)
def test_add_5(task):
"""Demonstrate ids."""
task_id = tasks.add(task)
t_from_db = tasks.get(task_id)
assert equivalent(t_from_db, task)
```

Let's run that and see how it looks:

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest -v test_add_variety.py::test_add_5
==================== test session starts ====================
collected 5 items

test_add_variety.py::test_add_5[Task(sleep,None,True)] PASSED
test_add_variety.py::test_add_5[Task(wake,brian,False)0] PASSED
test_add_variety.py::test_add_5[Task(wake,brian,False)1] PASSED
test_add_variety.py::test_add_5[Task(breathe,BRIAN,True)] PASSED
test_add_variety.py::test_add_5[Task(exercise,BrIaN,False)] PASSED

================== 5 passed in 0.04 seconds ==================
```

And these test identifiers can be used to run tests:

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest -v "test_add_variety.py::test_add_5[Task(exercise,BrIaN,False)]"
==================== test session starts ====================
collected 1 item

test_add_variety.py::test_add_5[Task(exercise,BrIaN,False)] PASSED

================== 1 passed in 0.03 seconds ==================
```

We definitely need quotes for these identifiers; otherwise, the brackets and parentheses will confuse the shell.

You can apply parametrize() to classes as well. When you do that, the same data sets will be sent to all test methods in the class:

ch2/tasks_proj/tests/func/test_add_variety.py

```
@pytest.mark.parametrize('task', tasks_to_try, ids=task_ids)
class TestAdd():
"""Demonstrate parametrize and test classes."""

def test_equivalent(self, task):
"""Similar test, just within a class."""
task_id = tasks.add(task)
t_from_db = tasks.get(task_id)
assert equivalent(t_from_db, task)
def test_valid_id(self, task):
"""We can use the same data or multiple tests."""
task_id = tasks.add(task)
t_from_db = tasks.get(task_id)
assert t_from_db.id == task_id
```

Here it is in action:

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest -v test_add_variety.py::TestAdd
==================== test session starts ====================
collected 10 items

test_add_variety.py::TestAdd::test_equivalent[Task(sleep,None,True)] PASSED
test_add_variety.py::TestAdd::test_equivalent[Task(wake,brian,False)0] PASSED
test_add_variety.py::TestAdd::test_equivalent[Task(wake,brian,False)1] PASSED
test_add_variety.py::TestAdd::test_equivalent[Task(breathe,BRIAN,True)] PASSED
test_add_variety.py::TestAdd::test_equivalent[Task(exercise,BrIaN,False)] PASSED
test_add_variety.py::TestAdd::test_valid_id[Task(sleep,None,True)] PASSED
test_add_variety.py::TestAdd::test_valid_id[Task(wake,brian,False)0] PASSED
test_add_variety.py::TestAdd::test_valid_id[Task(wake,brian,False)1] PASSED
test_add_variety.py::TestAdd::test_valid_id[Task(breathe,BRIAN,True)] PASSED
test_add_variety.py::TestAdd::test_valid_id[Task(exercise,BrIaN,False)] PASSED

================== 10 passed in 0.08 seconds ==================
```

You can also identify parameters by including an id right alongside the parameter value when passing in a list within the @pytest.mark.parametrize() decorator. You do this with pytest.param(<value>, id="something") syntax:

ch2/tasks_proj/tests/func/test_add_variety.py

```
@pytest.mark.parametrize('task', [
pytest.param(Task('create'), id='just summary'),
pytest.param(Task('inspire', 'Michelle'), id='summary/owner'),
pytest.param(Task('encourage', 'Michelle', True), id='summary/owner/done')])
def test_add_6(task):
"""Demonstrate pytest.param and id."""
task_id = tasks.add(task)
t_from_db = tasks.get(task_id)
assert equivalent(t_from_db, task)
```

In action:

```
$ cd /path/to/code/ch2/tasks_proj/tests/func
$ pytest -v test_add_variety.py::test_add_6
================== test session starts ==================
collected 3 items

test_add_variety.py::test_add_6[just summary] PASSED
test_add_variety.py::test_add_6[summary/owner] PASSED
test_add_variety.py::test_add_6[summary/owner/done] PASSED

================= 3 passed in 0.05 seconds =================
```

This is useful when the id cannot be derived from the parameter value.

Exercises

1. Download the project for this chapter, tasks_proj, from the book's webpage[7] and make sure you can install it locally with pip install /path/to/tasks_proj.
2. Explore the tests directory.
3. Run pytest with a single file.
4. Run pytest against a single directory, such as tasks_proj/tests/func. Use pytest to run tests individually as well as a directory full at a time. There are some failing tests there. Do you understand why they fail?
5. Add xfail or skip markers to the failing tests until you can run pytest from the tests directory with no arguments and no failures.
6. We don't have any tests for tasks.count() yet, among other functions. Pick an untested API function and think of which test cases we need to have to make sure it works correctly.
7. What happens if you try to add a task with the id already set? There are some missing exception tests in test_api_exceptions.py. See if you can fill in the missing exceptions. (It's okay to look at api.py for this exercise.)

What’s Next

You’ve run through a lot of the power of pytest in this chapter. Even with just what’s covered here, you can start supercharging your test suites. In many of the examples, you used a fixture called initialized_tasks_db. Fixtures can separate retrieving and/or generating test data from the real guts of a test function. They can also separate common code so that multiple test functions can use the same setup. In the next chapter, you’ll take a deep dive into the wonderful world of pytest fixtures.

Footnotes

[5]

https://pragprog.com/titles/bopytest/source_code

[6]

http://doc.pytest.org/en/latest/example/reportingdemo.html

[7]

https://pragprog.com/titles/bopytest/source_code

Chapter 3
pytest Fixtures

Now that you've seen the basics of pytest, let's turn our attention to fixtures, which are essential to structuring test code for almost any non-trivial software system. Fixtures are functions that are run by pytest before (and sometimes after) the actual test functions. The code in the fixture can do whatever you want it to. You can use fixtures to get a data set for the tests to work on. You can use fixtures to get a system into a known state before running a test. Fixtures are also used to get data ready for multiple tests.

Here's a simple fixture that returns a number:

ch3/test_fixtures.py

```
import pytest


@pytest.fixture()
def some_data():
"""Return answer to ultimate question."""
return 42


def test_some_data(some_data):
"""Use fixture return value in a test."""
assert some_data == 42
```

The @pytest.fixture() decorator is used to tell pytest that a function is a fixture. When you include the fixture name in the parameter list of a test function, pytest knows to run it before running the test. Fixtures can do work, and can also return data to the test function.

The test test_some_data() has the name of the fixture, some_data, as a parameter. pytest will see this and look for a fixture with this name. Naming is significant in pytest. pytest will look in the module of the test for a fixture of that name. It will also look in conftest.py files if it doesn't find it in this file.

Before we start our exploration of fixtures (and the conftest.py file), I need to address the fact that the term fixture has many meanings in the programming and test community, and even in the Python community. I use "fixture," "fixture function," and "fixture method" interchangeably to refer to the @pytest.fixture() decorated functions discussed in this chapter. Fixture can also be used to refer to the resource that is being set up by the fixture functions. Fixture functions often set up or retrieve some data that the test can work with. Sometimes this data is considered a fixture. For example, the Django community often uses fixture to mean some initial data that gets loaded into a database at the start of an application.

Regardless of other meanings, in pytest and in this book, test fixtures refer to the mechanism

pytest provides to allow the separation of “getting ready for” and “cleaning up after” code from your test functions.

pytest fixtures are one of the unique core features that make pytest stand out above other test frameworks, and are the reason why many people switch to and stay with pytest. However, fixtures in pytest are different than fixtures in Django and different than the setup and teardown procedures found in unittest and nose. There are a lot of features and nuances about fixtures. Once you get a good mental model of how they work, they will seem easy to you. However, you have to play with them a while to get there, so let’s get started.

Sharing Fixtures Through conftest.py

You can put fixtures into individual test files, but to share fixtures among multiple test files, you need to use a conftest.py file somewhere centrally located for all of the tests. For the Tasks project, all of the fixtures will be in tasks_proj/tests/conftest.py.

From there, the fixtures can be shared by any test. You can put fixtures in individual test files if you want the fixture to only be used by tests in that file. Likewise, you can have other conftest.py files in subdirectories of the top tests directory. If you do, fixtures defined in these lower-level conftest.py files will be available to tests in that directory and subdirectories. So far, however, the fixtures in the Tasks project are intended to be available to any test. Therefore, putting all of our fixtures in the conftest.py file at the test root, tasks_proj/tests, makes the most sense.

Although conftest.py is a Python module, it should not be imported by test files. Don't import conftest from anywhere. The conftest.py file gets read by pytest, and is considered a local plugin, which will make sense once we start talking about plugins in Chapter 5, *Plugins*. For now, think of tests/conftest.py as a place where we can put fixtures used by all tests under the tests directory.

Next, let's rework some our tests for tasks_proj to properly use fixtures.

Using Fixtures for Setup and Teardown

Most of the tests in the Tasks project will assume that the Tasks database is already set up and running and ready. And we should clean things up at the end if there is any cleanup needed. And maybe also disconnect from the database. Luckily, most of this is taken care of within the tasks code with tasks.start_tasks_db(<directory to store db>, 'tiny' or 'mongo') and tasks.stop_tasks_db(); we just need to call them at the right time, and we need a temporary directory.

Fortunately, pytest includes a cool fixture called tmpdir that we can use for testing and don't have to worry about cleaning up. It's not magic, just good coding by the pytest folks. (Don't worry; we look at tmpdir and it's session-scoped relative tmpdir_factory in more depth in *Using tmpdir and tmpdir_factory*.)

Given those pieces, this fixture works nicely:

ch3/a/tasks_proj/tests/conftest.py

```
import pytest
import tasks
from tasks import Task


@pytest.fixture()
def tasks_db(tmpdir):
"""Connect to db before tests, disconnect after."""
# Setup : start db
tasks.start_tasks_db(str(tmpdir), 'tiny')

yield # this is where the testing happens

# Teardown : stop db
tasks.stop_tasks_db()
```

The value of tmpdir isn't a string—it's an object that represents a directory. However, it implements __str__, so we can use str() to get a string to pass to start_tasks_db(). We're still using 'tiny' for TinyDB, for now.

A fixture function runs before the tests that use it. However, if there is a yield in the function, it stops there, passes control to the tests, and picks up on the next line after the tests are done. Therefore, think of the code above the yield as "setup" and the code after yield as "teardown." The code after the yield, the "teardown," is guaranteed to run regardless of what happens during the tests. We're not returning any data with the yield in this fixture. But you can.

Let's change one of our tasks.add() tests to use this fixture:

ch3/a/tasks_proj/tests/func/test_add.py

```
import pytest
import tasks
from tasks import Task


def test_add_returns_valid_id(tasks_db):
    """tasks.add(<valid task>) should return an integer."""
    # GIVEN an initialized tasks db
    # WHEN a new task is added
    # THEN returned task_id is of type int
    new_task = Task('do something')
    task_id = tasks.add(new_task)
    assert isinstance(task_id, int)
```

The main change here is that the extra fixture in the file has been removed, and we've added tasks_db to the parameter list of the test. I like to structure tests in a GIVEN/WHEN/THEN format using comments, especially when it isn't obvious from the code what's going on. I think it's helpful in this case. Hopefully, GIVEN an initialized tasks db helps to clarify why tasks_db is used as a fixture for the test.

Make Sure Tasks Is Installed

We're still writing tests to be run against the Tasks project in this chapter, which was first installed in Chapter 2. If you skipped that chapter, be sure to install tasks with cd code; pip install ./tasks_proj/.

Tracing Fixture Execution with –setup-show

If you run the test from the last section, you don't get to see what fixtures are run:

```
$ cd /path/to/code/
$ pip install ./tasks_proj/ # if not installed yet
$ cd /path/to/code/ch3/a/tasks_proj/tests/func
$ pytest -v test_add.py -k valid_id
==================== test session starts ====================
collected 3 items

test_add.py::test_add_returns_valid_id PASSED

==================== 2 tests deselected ====================
============ 1 passed, 2 deselected in 0.02 seconds ============
```

When I'm developing fixtures, I like to see what's running and when. Fortunately, pytest provides a command-line flag, --setup-show, that does just that:

```
$ pytest --setup-show test_add.py -k valid_id
==================== test session starts ====================
collected 3 items

test_add.py
SETUP S tmpdir_factory
SETUP F tmpdir (fixtures used: tmpdir_factory)
SETUP F tasks_db (fixtures used: tmpdir)
func/test_add.py::test_add_returns_valid_id
(fixtures used: tasks_db, tmpdir, tmpdir_factory).
TEARDOWN F tasks_db
TEARDOWN F tmpdir
TEARDOWN S tmpdir_factory

==================== 2 tests deselected ====================
============ 1 passed, 2 deselected in 0.02 seconds ============
```

Our test is in the middle, and pytest designates a SETUP and TEARDOWN portion to each fixture. Going from test_add_returns_valid_id up, you see that tmpdir ran before the test. And before that, tmpdir_factory. Apparently, tmpdir uses it as a fixture.

The F and S in front of the fixture names indicate scope. F for function scope, and S for session scope. I'll talk about scope in *Specifying Fixture Scope*.

Using Fixtures for Test Data

Fixtures are a great place to store data to use for testing. You can return anything. Here's a fixture returning a tuple of mixed type:

ch3/test_fixtures.py

```
@pytest.fixture()
def a_tuple():
"""Return something more interesting."""
return (1, 'foo', None, {'bar': 23})


def test_a_tuple(a_tuple):
"""Demo the a_tuple fixture."""
assert a_tuple[3]['bar'] == 32
```

Since test_a_tuple() should fail (23 != 32), we can see what happens when a test with a fixture fails:

```
$ cd /path/to/code/ch3
$ pytest test_fixtures.py::test_a_tuple
==================== test session starts ====================
collected 1 item

test_fixtures.py F


============================ FAILURES ============================
_____________________________ test_a_tuple _____________________________

a_tuple = (1, 'foo', None, {'bar': 23})

def test_a_tuple(a_tuple):
"""Demo the a_tuple fixture."""
> assert a_tuple[3]['bar'] == 32
E assert 23 == 32

test_fixtures.py:43: AssertionError
================== 1 failed in 0.07 seconds ==================
```

Along with the stack trace section, pytest reports the value parameters of the function that raised the exception or failed an assert. In the case of tests, the fixtures are parameters to the test, and

are therefore reported with the stack trace.

What happens if the assert (or any exception) happens in the fixture?

```
$ pytest -v test_fixtures.py::test_other_data
==================== test session starts ====================
collected 1 item

test_fixtures.py::test_other_data ERROR

============================ ERRORS ============================
_______________ ERROR at setup of test_other_data _______________

@pytest.fixture()
def some_other_data():
"""Raise an exception from fixture."""
x = 43
> assert x == 42
E assert 43 == 42

test_fixtures.py:24: AssertionError
=================== 1 error in 0.04 seconds ====================
```

A couple of things happen. The stack trace shows correctly that the assert happened in the fixture function. Also, test_other_data is reported not as FAIL, but as ERROR. This distinction is great. If a test ever fails, you know the failure happened in the test proper, and not in any fixture it depends on.

But what about the Tasks project? For the Tasks project, we could probably use some data fixtures, perhaps different lists of tasks with various properties:

ch3/a/tasks_proj/tests/conftest.py

```
# Reminder of Task constructor interface
# Task(summary=None, owner=None, done=False, id=None)
# summary is required
# owner and done are optional
# id is set by database

@pytest.fixture()
def tasks_just_a_few():
"""All summaries and owners are unique."""
return (
Task('Write some code', 'Brian', True),
Task("Code review Brian's code", 'Katie', False),
```

```
    Task('Fix what Brian did', 'Michelle', False))


@pytest.fixture()
def tasks_mult_per_owner():
"""Several owners with several tasks each."""
return (
Task('Make a cookie', 'Raphael'),
Task('Use an emoji', 'Raphael'),
Task('Move to Berlin', 'Raphael'),

Task('Create', 'Michelle'),
Task('Inspire', 'Michelle'),
Task('Encourage', 'Michelle'),

Task('Do a handstand', 'Daniel'),
Task('Write some books', 'Daniel'),
Task('Eat ice cream', 'Daniel'))
```

You can use these directly from tests, or you can use them from other fixtures. Let's use them to build up some non-empty databases to use for testing.

Using Multiple Fixtures

You've already seen that tmpdir uses tmpdir_factory. And you used tmpdir in our tasks_db fixture. Let's keep the chain going and add some specialized fixtures for non-empty tasks databases:

ch3/a/tasks_proj/tests/conftest.py

```
@pytest.fixture()
def db_with_3_tasks(tasks_db, tasks_just_a_few):
"""Connected db with 3 tasks, all unique."""
for t in tasks_just_a_few:
tasks.add(t)


@pytest.fixture()
def db_with_multi_per_owner(tasks_db, tasks_mult_per_owner):
"""Connected db with 9 tasks, 3 owners, all with 3 tasks."""
for t in tasks_mult_per_owner:
tasks.add(t)
```

These fixtures all include two fixtures each in their parameter list: tasks_db and a data set. The data set is used to add tasks to the database. Now tests can use these when you want the test to start from a non-empty database, like this:

ch3/a/tasks_proj/tests/func/test_add.py

```
def test_add_increases_count(db_with_3_tasks):
"""Test tasks.add() affect on tasks.count()."""
# GIVEN a db with 3 tasks
# WHEN another task is added
tasks.add(Task('throw a party'))

# THEN the count increases by 1
assert tasks.count() == 4
```

This also demonstrates one of the great reasons to use fixtures: to focus the test on what you're actually testing, not on what you had to do to get ready for the test. I like using comments for GIVEN/WHEN/THEN and trying to push as much GIVEN into fixtures for two reasons. First, it makes the test more readable and, therefore, more maintainable. Second, an assert or exception in the fixture results in an ERROR, while an assert or exception in a test function results in a FAIL. I don't want test_add_increases_count() to FAIL if database initialization failed. That would just be confusing. I want a FAIL for test_add_increases_count() to only be possible if add() really failed to alter the count. Let's trace it and see all the fixtures run:

```
$ cd /path/to/code/ch3/a/tasks_proj/tests/func
```

```
$ pytest --setup-show test_add.py::test_add_increases_count
========== test session starts ==========
collected 1 item

test_add.py
SETUP S tmpdir_factory
SETUP F tmpdir (fixtures used: tmpdir_factory)
SETUP F tasks_db (fixtures used: tmpdir)
SETUP F tasks_just_a_few
SETUP F db_with_3_tasks (fixtures used: tasks_db, tasks_just_a_few)
func/test_add.py::test_add_increases_count
(fixtures used: db_with_3_tasks, tasks_db, tasks_just_a_few,
tmpdir, tmpdir_factory).
TEARDOWN F db_with_3_tasks
TEARDOWN F tasks_just_a_few
TEARDOWN F tasks_db
TEARDOWN F tmpdir
TEARDOWN S tmpdir_factory

========== 1 passed in 0.04 seconds ==========
```

There are those F's and S's for function and session scope again. Let's learn about those next.

Specifying Fixture Scope

Fixtures include an optional parameter called scope, which controls how often a fixture gets set up and torn down. The scope parameter to @pytest.fixture() can have the values of function, class, module, or session. The default scope is function. The tasks_db fixture and all of the fixtures so far don't specify a scope. Therefore, they are function scope fixtures.

Here's a rundown of each scope value:

scope='function'

> Run once per test function. The setup portion is run before each test using the fixture. The teardown portion is run after each test using the fixture. This is the default scope used when no scope parameter is specified.

scope='class'

> Run once per test class, regardless of how many test methods are in the class.

scope='module'

> Run once per module, regardless of how many test functions or methods or other fixtures in the module use it.

scope='session'

> Run once per session. All test methods and functions using a fixture of session scope share one setup and teardown call.

Here's how the scope values look in action:

ch3/test_scope.py

```
"""Demo fixture scope."""

import pytest


@pytest.fixture(scope='function')
def func_scope():
"""A function scope fixture."""


@pytest.fixture(scope='module')
def mod_scope():
"""A module scope fixture."""
```

```
@pytest.fixture(scope='session')
def sess_scope():
"""A session scope fixture."""


@pytest.fixture(scope='class')
def class_scope():
"""A class scope fixture."""


def test_1(sess_scope, mod_scope, func_scope):
"""Test using session, module, and function scope fixtures."""


def test_2(sess_scope, mod_scope, func_scope):
"""Demo is more fun with multiple tests."""

@pytest.mark.usefixtures('class_scope')
class TestSomething():
"""Demo class scope fixtures."""

def test_3(self):
"""Test using a class scope fixture."""

def test_4(self):
"""Again, multiple tests are more fun."""
```

Let's use --setup-show to demonstrate that the number of times a fixture is called and when the setup and teardown are run depend on the scope:

```
$ cd /path/to/code/ch3
$ pytest --setup-show test_scope.py
======================== test session starts ========================
collected 4 items

test_scope.py
SETUP S sess_scope
SETUP M mod_scope
SETUP F func_scope
test_scope.py::test_1
(fixtures used: func_scope, mod_scope, sess_scope).
TEARDOWN F func_scope
```

```
SETUP F func_scope
test_scope.py::test_2
(fixtures used: func_scope, mod_scope, sess_scope).
TEARDOWN F func_scope
SETUP C class_scope
test_scope.py::TestSomething::()::test_3 (fixtures used: class_scope).
test_scope.py::TestSomething::()::test_4 (fixtures used: class_scope).
TEARDOWN C class_scope
TEARDOWN M mod_scope
TEARDOWN S sess_scope

==================== 4 passed in 0.01 seconds ====================
```

Now you get to see not just F and S for function and session, but also C and M for class and module.

Scope is defined with the fixture. I know this is obvious from the code, but it's an important point to make sure you fully grok. The scope is set at the definition of a fixture, and not at the place where it's called. The test functions that use a fixture don't control how often a fixture is set up and torn down.

Fixtures can only depend on other fixtures of their same scope or wider. So a function scope fixture can depend on other function scope fixtures (the default, and used in the Tasks project so far). A function scope fixture can also depend on class, module, and session scope fixtures, but you can't go in the reverse order.

Changing Scope for Tasks Project Fixtures

With this knowledge of scope, let's now change the scope of some of the Task project fixtures.

So far, we haven't had a problem with test times. But it seems like a waste to set up a temporary directory and new connection to a database for every test. As long as we can ensure an empty database when needed, that should be sufficient.

To have something like tasks_db be session scope, you need to use tmpdir_factory, since tmpdir is function scope and tmpdir_factory is session scope. Luckily, this is just a one-line code change (well, two if you count tmpdir -> tmpdir_factory in the parameter list):

ch3/b/tasks_proj/tests/conftest.py

```
import pytest
import tasks
from tasks import Task


@pytest.fixture(scope='session')
def tasks_db_session(tmpdir_factory):
"""Connect to db before tests, disconnect after."""
```

```
    temp_dir = tmpdir_factory.mktemp('temp')
    tasks.start_tasks_db(str(temp_dir), 'tiny')
    yield
    tasks.stop_tasks_db()


@pytest.fixture()
def tasks_db(tasks_db_session):
    """An empty tasks db."""
    tasks.delete_all()
```

Here we changed tasks_db to depend on tasks_db_session, and we deleted all the entries to make sure it's empty. Because we didn't change its name, none of the fixtures or tests that already include it have to change.

The data fixtures just return a value, so there really is no reason to have them run all the time. Once per session is sufficient:

ch3/b/tasks_proj/tests/conftest.py

```
# Reminder of Task constructor interface
# Task(summary=None, owner=None, done=False, id=None)
# summary is required
# owner and done are optional
# id is set by database
@pytest.fixture(scope='session')
def tasks_just_a_few():
    """All summaries and owners are unique."""
    return (
        Task('Write some code', 'Brian', True),
        Task("Code review Brian's code", 'Katie', False),
        Task('Fix what Brian did', 'Michelle', False))


@pytest.fixture(scope='session')
def tasks_mult_per_owner():
    """Several owners with several tasks each."""
    return (
        Task('Make a cookie', 'Raphael'),
        Task('Use an emoji', 'Raphael'),
        Task('Move to Berlin', 'Raphael'),

        Task('Create', 'Michelle'),
        Task('Inspire', 'Michelle'),
```

```
    Task('Encourage', 'Michelle'),

    Task('Do a handstand', 'Daniel'),
    Task('Write some books', 'Daniel'),
    Task('Eat ice cream', 'Daniel'))
```

Now, let's see if all of these changes work with our tests:

```
$ cd /path/to/code/ch3/b/tasks_proj
$ pytest
==================== test session starts ====================
collected 55 items

tests/func/test_add.py ...
tests/func/test_add_variety.py .........................
tests/func/test_add_variety2.py ............
tests/func/test_api_exceptions.py .......
tests/func/test_unique_id.py .
tests/unit/test_task.py ....

=================== 55 passed in 0.17 seconds ===================
```

Looks like it's all good. Let's trace the fixtures for one test file to see if the different scoping worked as expected:

```
$ pytest --setup-show tests/func/test_add.py
======================== test session starts ========================
collected 3 items

tests/func/test_add.py
SETUP S tmpdir_factory
SETUP S tasks_db_session (fixtures used: tmpdir_factory)
SETUP F tasks_db (fixtures used: tasks_db_session)
tests/func/test_add.py::test_add_returns_valid_id
(fixtures used: tasks_db, tasks_db_session, tmpdir_factory).
TEARDOWN F tasks_db
SETUP F tasks_db (fixtures used: tasks_db_session)
tests/func/test_add.py::test_added_task_has_id_set
(fixtures used: tasks_db, tasks_db_session, tmpdir_factory).
TEARDOWN F tasks_db
SETUP F tasks_db (fixtures used: tasks_db_session)
SETUP S tasks_just_a_few
SETUP F db_with_3_tasks (fixtures used: tasks_db, tasks_just_a_few)
```

```
  tests/func/test_add.py::test_add_increases_count
  (fixtures used: db_with_3_tasks, tasks_db, tasks_db_session,
  tasks_just_a_few, tmpdir_factory).
  TEARDOWN F db_with_3_tasks
  TEARDOWN F tasks_db
  TEARDOWN S tasks_just_a_few
  TEARDOWN S tasks_db_session
  TEARDOWN S tmpdir_factory

  ====================== 3 passed in 0.03 seconds ======================
```

Yep. Looks right. tasks_db_session is called once per session, and the quicker tasks_db now just cleans out the database before each test.

Specifying Fixtures with usefixtures

So far, if you wanted a test to use a fixture, you put it in the parameter list. You can also mark a test or a class with @pytest.mark.usefixtures('fixture1', 'fixture2'). usefixtures takes a string that is composed of a comma-separated list of fixtures to use. It doesn't make sense to do this with test functions—it's just more typing. But it does work well for test classes:

ch3/test_scope.py

```
@pytest.mark.usefixtures('class_scope')
class TestSomething():
    """Demo class scope fixtures."""

    def test_3(self):
        """Test using a class scope fixture."""

    def test_4(self):
        """Again, multiple tests are more fun."""
```

Using usefixtures is almost the same as specifying the fixture name in the test method parameter list. The one difference is that the test can use the return value of a fixture only if it's specified in the parameter list. A test using a fixture due to usefixtures cannot use the fixture's return value.

Using autouse for Fixtures That Always Get Used

So far in this chapter, all of the fixtures used by tests were named by the tests (or used usefixtures for that one class example). However, you can use autouse=True to get a fixture to run all of the time. This works well for code you want to run at certain times, but tests don't really depend on any system state or data from the fixture. Here's a rather contrived example:

ch3/test_autouse.py

```
"""Demonstrate autouse fixtures."""

import pytest
import time


@pytest.fixture(autouse=True, scope='session')
def footer_session_scope():
"""Report the time at the end of a session."""
yield
now = time.time()
print('--')
print('finished : {}'.format(time.strftime('%d %b %X', time.localtime(now))))
print('-----------------')


@pytest.fixture(autouse=True)
def footer_function_scope():
"""Report test durations after each function."""
start = time.time()
yield
stop = time.time()
delta = stop - start
print('\ntest duration : {:0.3} seconds'.format(delta))


def test_1():
"""Simulate long-ish running test."""
time.sleep(1)


def test_2():
"""Simulate slightly longer test."""
```

```
  time.sleep(1.23)
```

We want to add test times after each test, and the date and current time at the end of the session. Here's what these look like:

```
  $ cd /path/to/code/ch3
  $ pytest -v -s test_autouse.py
  ===================== test session starts ======================
  collected 2 items

  test_autouse.py::test_1 PASSED
  test duration : 1.0 seconds

  test_autouse.py::test_2 PASSED
  test duration : 1.24 seconds
  --
  finished : 25 Jul 16:18:27
  -----------------
  =================== 2 passed in 2.25 seconds ===================
```

The autouse feature is good to have around. But it's more of an exception than a rule. Opt for named fixtures unless you have a really great reason not to.

Now that you've seen autouse in action, you may be wondering why we didn't use it for tasks_db in this chapter. In the Tasks project, I felt it was important to keep the ability to test what happens if we try to use an API function before db initialization. It should raise an appropriate exception. But we can't test this if we force good initialization on every test.

Renaming Fixtures

The name of a fixture, listed in the parameter list of tests and other fixtures using it, is usually the same as the function name of the fixture. However, pytest allows you to rename fixtures with a name parameter to @pytest.fixture():

ch3/test_rename_fixture.py

```
"""Demonstrate fixture renaming."""

import pytest


@pytest.fixture(name='lue')
def ultimate_answer_to_life_the_universe_and_everything():
"""Return ultimate answer."""
return 42


def test_everything(lue):
"""Use the shorter name."""
assert lue == 42
```

Here, lue is now the fixture name, instead of fixture_with_a_name_much_longer_than_lue. That name even shows up if we run it with --setup-show:

```
$ pytest --setup-show test_rename_fixture.py
======================== test session starts ========================
collected 1 items

test_rename_fixture.py
SETUP F lue
test_rename_fixture.py::test_everything_2 (fixtures used: lue).
TEARDOWN F lue

===================== 1 passed in 0.01 seconds ======================
```

If you need to find out where lue is defined, you can add the pytest option --fixtures and give it the filename for the test. It lists all the fixtures available for the test, including ones that have been renamed:

```
$ pytest --fixtures test_rename_fixture.py
======================== test session starts ========================
...
```

---------- fixtures defined from test_rename_fixture ----------
lue
Return ultimate answer.

================= no tests ran in 0.01 seconds =================

Most of the output is omitted—there's a lot there. Luckily, the fixtures we defined are at the bottom, along with where they are defined. We can use this to look up the definition of lue. Let's use that in the Tasks project:

```
$ cd /path/to/code/ch3/b/tasks_proj
$ pytest --fixtures tests/func/test_add.py
======================== test session starts ========================
...
tmpdir_factory
Return a TempdirFactory instance for the test session.
tmpdir
Return a temporary directory path object which is
unique to each test function invocation, created as
a sub directory of the base temporary directory.
The returned object is a `py.path.local`_path object.

------------------ fixtures defined from conftest -------------------
tasks_db_session
Connect to db before tests, disconnect after.
tasks_db
An empty tasks db.
tasks_just_a_few
All summaries and owners are unique.
tasks_mult_per_owner
Several owners with several tasks each.
db_with_3_tasks
Connected db with 3 tasks, all unique.
db_with_multi_per_owner
Connected db with 9 tasks, 3 owners, all with 3 tasks.

=================== no tests ran in 0.01 seconds ====================
```

Cool. All of our conftest.py fixtures are there. And at the bottom of the builtin list is the tmpdir and tmpdir_factory that we used also.

Parametrizing Fixtures

In *Parametrized Testing*, we parametrized tests. We can also parametrize fixtures. We still use our list of tasks, list of task identifiers, and an equivalence function, just as before:

ch3/b/tasks_proj/tests/func/test_add_variety2.py

```
import pytest
import tasks
from tasks import Task

tasks_to_try = (Task('sleep', done=True),
Task('wake', 'brian'),
Task('breathe', 'BRIAN', True),
Task('exercise', 'BrIaN', False))

task_ids = ['Task({},{},{})'.format(t.summary, t.owner, t.done)
for t in tasks_to_try]


def equivalent(t1, t2):
"""Check two tasks for equivalence."""
return ((t1.summary == t2.summary) and
(t1.owner == t2.owner) and
(t1.done == t2.done))
```

But now, instead of parametrizing the test, we will parametrize a fixture called a_task:

ch3/b/tasks_proj/tests/func/test_add_variety2.py

```
@pytest.fixture(params=tasks_to_try)
def a_task(request):
"""Using no ids."""
return request.param


def test_add_a(tasks_db, a_task):
"""Using a_task fixture (no ids)."""
task_id = tasks.add(a_task)
t_from_db = tasks.get(task_id)
assert equivalent(t_from_db, a_task)
```

The request listed in the fixture parameter is another builtin fixture that represents the calling state of the fixture. You'll explore it more in the next chapter. It has a field param that is filled in

with one element from the list assigned to params in @pytest.fixture(params=tasks_to_try).

The a_task fixture is pretty simple—it just returns the request.param as its value to the test using it. Since our task list has four tasks, the fixture will be called four times, and then the test will get called four times:

```
$ cd /path/to/code/ch3/b/tasks_proj/tests/func
$ pytest -v test_add_variety2.py::test_add_a
==================== test session starts ====================
collected 4 items

test_add_variety2.py::test_add_a[a_task0] PASSED
test_add_variety2.py::test_add_a[a_task1] PASSED
test_add_variety2.py::test_add_a[a_task2] PASSED
test_add_variety2.py::test_add_a[a_task3] PASSED

=================== 4 passed in 0.03 seconds ===================
```

We didn't provide ids, so pytest just made up some names by appending a number to the name of the fixture. However, we can use the same string list we used when we parametrized our tests:

ch3/b/tasks_proj/tests/func/test_add_variety2.py

```
@pytest.fixture(params=tasks_to_try, ids=task_ids)
def b_task(request):
"""Using a list of ids."""
return request.param


def test_add_b(tasks_db, b_task):
"""Using b_task fixture, with ids."""
task_id = tasks.add(b_task)
t_from_db = tasks.get(task_id)
assert equivalent(t_from_db, b_task)
```

This gives us better identifiers:

```
$ pytest -v test_add_variety2.py::test_add_b
==================== test session starts ====================
collected 4 items

test_add_variety2.py::test_add_b[Task(sleep,None,True)] PASSED
test_add_variety2.py::test_add_b[Task(wake,brian,False)] PASSED
test_add_variety2.py::test_add_b[Task(breathe,BRIAN,True)] PASSED
test_add_variety2.py::test_add_b[Task(exercise,BrIaN,False)] PASSED
```

```
================== 4 passed in 0.04 seconds ==================
```

We can also set the ids parameter to a function we write that provides the identifiers. Here's what it looks like when we use a function to generate the identifiers:

ch3/b/tasks_proj/tests/func/test_add_variety2.py

```
def id_func(fixture_value):
"""A function for generating ids."""
t = fixture_value
return 'Task({},{},{})'.format(t.summary, t.owner, t.done)


@pytest.fixture(params=tasks_to_try, ids=id_func)
def c_task(request):
"""Using a function (id_func) to generate ids."""
return request.param


def test_add_c(tasks_db, c_task):
"""Use fixture with generated ids."""
task_id = tasks.add(c_task)
t_from_db = tasks.get(task_id)
assert equivalent(t_from_db, c_task)
```

The function will be called from the value of each item from the parametrization. Since the parametrization is a list of Task objects, id_func() will be called with a Task object, which allows us to use the namedtuple accessor methods to access a single Task object to generate the identifier for one Task object at a time. It's a bit cleaner than generating a full list ahead of time, and looks the same:

```
$ pytest -v test_add_variety2.py::test_add_c
==================== test session starts ====================
collected 4 items

test_add_variety2.py::test_add_c[Task(sleep,None,True)] PASSED
test_add_variety2.py::test_add_c[Task(wake,brian,False)] PASSED
test_add_variety2.py::test_add_c[Task(breathe,BRIAN,True)] PASSED
test_add_variety2.py::test_add_c[Task(exercise,BrIaN,False)] PASSED

================== 4 passed in 0.04 seconds ==================
```

With parametrized functions, you get to run that function multiple times. But with parametrized fixtures, every test function that uses that fixture will be called multiple times. Very powerful.

Parametrizing Fixtures in the Tasks Project

Now, let's see how we can use parametrized fixtures in the Tasks project. So far, we used TinyDB for all of the testing. But we want to keep our options open until later in the project. Therefore, any code we write, and any tests we write, should work with both TinyDB and with MongoDB.

The decision (in the code) of which database to use is isolated to the start_tasks_db() call in the tasks_db_session fixture:

ch3/b/tasks_proj/tests/conftest.py

```
import pytest
import tasks
from tasks import Task


@pytest.fixture(scope='session')
def tasks_db_session(tmpdir_factory):
"""Connect to db before tests, disconnect after."""
temp_dir = tmpdir_factory.mktemp('temp')
tasks.start_tasks_db(str(temp_dir), 'tiny')
yield
tasks.stop_tasks_db()


@pytest.fixture()
def tasks_db(tasks_db_session):
"""An empty tasks db."""
tasks.delete_all()
```

The db_type parameter in the call to start_tasks_db() isn't magic. It just ends up switching which subsystem gets to be responsible for the rest of the database interactions:

tasks_proj/src/tasks/api.py

```
def start_tasks_db(db_path, db_type): # type: (str, str) -> None
"""Connect API functions to a db."""
if not isinstance(db_path, string_types):
raise TypeError('db_path must be a string')
global _tasksdb
if db_type == 'tiny':
import tasks.tasksdb_tinydb
_tasksdb = tasks.tasksdb_tinydb.start_tasks_db(db_path)
elif db_type == 'mongo':
import tasks.tasksdb_pymongo
```

```
    _tasksdb = tasks.tasksdb_pymongo.start_tasks_db(db_path)
else:
raise ValueError("db_type must be a 'tiny' or 'mongo'")
```

To test MongoDB, we need to run all the tests with db_type set to mongo. A small change does the trick:

ch3/c/tasks_proj/tests/conftest.py

```
import pytest
import tasks
from tasks import Task


#@pytest.fixture(scope='session', params=['tiny',])
@pytest.fixture(scope='session', params=['tiny', 'mongo'])
def tasks_db_session(tmpdir_factory, request):
"""Connect to db before tests, disconnect after."""
temp_dir = tmpdir_factory.mktemp('temp')
tasks.start_tasks_db(str(temp_dir), request.param)
yield # this is where the testing happens
tasks.stop_tasks_db()


@pytest.fixture()
def tasks_db(tasks_db_session):
"""An empty tasks db."""
tasks.delete_all()
```

Here I added params=['tiny','mongo'] to the fixture decorator. I added request to the parameter list of temp_db, and I set db_type to request.param instead of just picking 'tiny' or 'mongo'.

When you set the --verbose or -v flag with pytest running parametrized tests or parametrized fixtures, pytest labels the different runs based on the value of the parametrization. And because the values are already strings, that works great.

Installing MongoDB

To follow along with MongoDB testing, make sure MongoDB and pymongo are installed. I've been testing with the community edition of MongoDB, found at https://www.mongodb.com/download-center. pymongo is installed with pip—pip install pymongo. However, using MongoDB is not necessary to follow along with the rest of the book; it's used in this example and in a debugger example in Chapter 7.

Here's what we have so far:

```
$ cd /path/to/code/ch3/c/tasks_proj
$ pip install pymongo
$ pytest -v --tb=no
==================== test session starts ====================
collected 92 items

test_add.py::test_add_returns_valid_id[tiny] PASSED
test_add.py::test_added_task_has_id_set[tiny] PASSED
test_add.py::test_add_increases_count[tiny] PASSED
test_add_variety.py::test_add_1[tiny] PASSED
test_add_variety.py::test_add_2[tiny-task0] PASSED
test_add_variety.py::test_add_2[tiny-task1] PASSED
...
test_add.py::test_add_returns_valid_id[mongo] FAILED
test_add.py::test_added_task_has_id_set[mongo] FAILED
test_add.py::test_add_increases_count[mongo] PASSED
test_add_variety.py::test_add_1[mongo] FAILED
test_add_variety.py::test_add_2[mongo-task0] FAILED
...
============= 42 failed, 50 passed in 4.94 seconds =============
```

Hmm. Bummer. Looks like we'll need to do some debugging before we let anyone use the Mongo version. You'll take a look at how to debug this in *pdb: Debugging Test Failures*. Until then, we'll use the TinyDB version.

Exercises

1. Create a test file called test_fixtures.py.
2. Write a few data fixtures—functions with the @pytest.fixture() decorator—that return some data. Perhaps a list, or a dictionary, or a tuple.
3. For each fixture, write at least one test function that uses it.
4. Write two tests that use the same fixture.
5. Run pytest --setup-show test_fixtures.py. Are all the fixtures run before every test?
6. Add scope='module' to the fixture from Exercise 4.
7. Re-run pytest --setup-show test_fixtures.py. What changed?
8. For the fixture from Exercise 6, change return <data> to yield <data>.
9. Add print statements before and after the yield.
10. Run pytest -s -v test_fixtures.py. Does the output make sense?

What’s Next

The pytest fixture implementation is flexible enough to use fixtures like building blocks to build up test setup and teardown, and to swap in and out different chunks of the system (like swapping in Mongo for TinyDB). Because fixtures are so flexible, I use them heavily to push as much of the setup of my tests into fixtures as I can.

In this chapter, you looked at pytest fixtures you write yourself, as well as a couple of builtin fixtures, tmpdir and tmpdir_factory. You’ll take a closer look at the builtin fixtures in the next chapter.

Chapter 4
Builtin Fixtures

In the previous chapter, you looked at what fixtures are, how to write them, and how to use them for test data as well as setup and teardown code. You also used conftest.py for sharing fixtures between tests in multiple test files. By the end of Chapter 3, *pytest Fixtures*, the Tasks project had these fixtures: tasks_db_session, tasks_just_a_few, tasks_mult_per_owner, tasks_db, db_with_3_tasks, and db_with_multi_per_owner defined in conftest.py to be used by any test function in the Tasks project that needed them.

Reusing common fixtures is such a good idea that the pytest developers included some commonly needed fixtures with pytest. You've already seen tmpdir and tmpdir_factory in use by the Tasks project in *Changing Scope for Tasks Project Fixtures*. You'll take a look at them in more detail in this chapter.

The builtin fixtures that come prepackaged with pytest can help you do some pretty useful things in your tests easily and consistently. For example, in addition to handling temporary files, pytest includes builtin fixtures to access command-line options, communicate between tests sessions, validate output streams, modify environmental variables, and interrogate warnings. The builtin fixtures are extensions to the core functionality of pytest. Let's now take a look at several of the most often used builtin fixtures one by one.

Using tmpdir and tmpdir_factory

The tmpdir and tmpdir_factory builtin fixtures are used to create a temporary file system directory before your test runs, and remove the directory when your test is finished. In the Tasks project, we needed a directory to store the temporary database files used by MongoDB and TinyDB. However, because we want to test with temporary databases that don't survive past a test session, we used tmpdir and tmpdir_factory to do the directory creation and cleanup for us.

If you're testing something that reads, writes, or modifies files, you can use tmpdir to create files or directories used by a single test, and you can use tmpdir_factory when you want to set up a directory for many tests.

The tmpdir fixture has function scope, and the tmpdir_factory fixture has session scope. Any individual test that needs a temporary directory or file just for the single test can use tmpdir. This is also true for a fixture that is setting up a directory or file that should be recreated for each test function.

Here's a simple example using tmpdir:

ch4/test_tmpdir.py

```
def test_tmpdir(tmpdir):
# tmpdir already has a path name associated with it
# join() extends the path to include a filename
# the file is created when it's written to
a_file = tmpdir.join('something.txt')

# you can create directories
a_sub_dir = tmpdir.mkdir('anything')

# you can create files in directories (created when written)
another_file = a_sub_dir.join('something_else.txt')

# this write creates 'something.txt'
a_file.write('contents may settle during shipping')

# this write creates 'anything/something_else.txt'
another_file.write('something different')

# you can read the files as well
assert a_file.read() == 'contents may settle during shipping'
assert another_file.read() == 'something different'
```

The value returned from tmpdir is an object of type py.path.local.[8] This seems like everything we need for temporary directories and files. However, there's one gotcha. Because the tmpdir

fixture is defined as function scope, you can't use tmpdir to create folders or files that should stay in place longer than one test function. For fixtures with scope other than function (class, module, session), tmpdir_factory is available.

The tmpdir_factory fixture is a lot like tmpdir, but it has a different interface. As discussed in *Specifying Fixture Scope*, function scope fixtures run once per test function, module scope fixtures run once per module, class scope fixtures run once per class, and test scope fixtures run once per session. Therefore, resources created in session scope fixtures have a lifetime of the entire session.

To see how similar tmpdir and tmpdir_factory are, I'll modify the tmpdir example just enough to use tmpdir_factory instead:

ch4/test_tmpdir.py

```
def test_tmpdir_factory(tmpdir_factory):
# you should start with making a directory
# a_dir acts like the object returned from the tmpdir fixture
a_dir = tmpdir_factory.mktemp('mydir')

# base_temp will be the parent dir of 'mydir'
# you don't have to use getbasetemp()
# using it here just to show that it's available
base_temp = tmpdir_factory.getbasetemp()
print('base:', base_temp)

# the rest of this test looks the same as the 'test_tmpdir()'
# example except I'm using a_dir instead of tmpdir

a_file = a_dir.join('something.txt')
a_sub_dir = a_dir.mkdir('anything')
another_file = a_sub_dir.join('something_else.txt')

a_file.write('contents may settle during shipping')
another_file.write('something different')

assert a_file.read() == 'contents may settle during shipping'
assert another_file.read() == 'something different'
```

The first line uses mktemp('mydir') to create a directory and saves it in a_dir. For the rest of the function, you can use a_dir just like the tmpdir returned from the tmpdir fixture.

In the second line of the tmpdir_factory example, the getbasetemp() function returns the base directory used for this session. The print statement is in the example so you can see where the directory is on your system. Let's see where it is:

```
$ cd /path/to/code/ch4
```

```
$ pytest -q -s test_tmpdir.py::test_tmpdir_factory
base: /private/var/folders/53/zv4j_zc506x2xq25l31qxvxm0000gn\
/T/pytest-of-okken/pytest-732
.
1 passed in 0.04 seconds
```

This base directory is system- and user-dependent, and pytest-NUM changes with an incremented NUM for every session. The base directory is left alone after a session, but pytest cleans them up and only the most recent few temporary base directories are left on the system, which is great if you need to inspect the files after a test run.

You can also specify your own base directory if you need to with pytest --basetemp=mydir.

Using Temporary Directories for Other Scopes

We get session scope temporary directories and files from the tmpdir_factory fixture, and function scope directories and files from the tmpdir fixture. But what about other scopes? What if we need a module or a class scope temporary directory? To do this, we create another fixture of the scope we want and have it use tmpdir_factory.

For example, suppose we have a module full of tests, and many of them need to be able to read some data from a json file. We could put a module scope fixture in either the module itself, or in a conftest.py file that sets up the data file like this:

ch4/authors/conftest.py

```
"""Demonstrate tmpdir_factory."""

import json
import pytest


@pytest.fixture(scope='module')
def author_file_json(tmpdir_factory):
"""Write some authors to a data file."""
python_author_data = {
'Ned': {'City': 'Boston'},
'Brian': {'City': 'Portland'},
'Luciano': {'City': 'Sau Paulo'}
}

file = tmpdir_factory.mktemp('data').join('author_file.json')
print('file:{}'.format(str(file)))

with file.open('w') as f:
json.dump(python_author_data, f)
```

```
    return file
```

The author_file_json() fixture creates a temporary directory called data and creates a file called author_file.json within the data directory. It then writes the python_author_data dictionary as json. Because this is a module scope fixture, the json file will only be created once per module that has a test using it:

ch4/authors/test_authors.py

```
"""Some tests that use temp data files."""
import json


def test_brian_in_portland(author_file_json):
"""A test that uses a data file."""
with author_file_json.open() as f:
authors = json.load(f)
assert authors['Brian']['City'] == 'Portland'


def test_all_have_cities(author_file_json):
"""Same file is used for both tests."""
with author_file_json.open() as f:
authors = json.load(f)
for a in authors:
assert len(authors[a]['City']) > 0
```

Both tests will use the same json file. If one test data file works for multiple tests, there's no use recreating it for both.

Using pytestconfig

With the pytestconfig builtin fixture, you can control how pytest runs through command-line arguments and options, configuration files, plugins, and the directory from which you launched pytest. The pytestconfig fixture is a shortcut to request.config, and is sometimes referred to in the pytest documentation as “the pytest config object.”

To see how pytestconfig works, you’ll look at how to add a custom command-line option and read the option value from within a test. You can read the value of command-line options directly from pytestconfig, but to add the option and have pytest parse it, you need to add a hook function. Hook functions, which I cover in more detail in Chapter 5, *Plugins*, are another way to control how pytest behaves and are used frequently in plugins. However, adding a custom command-line option and reading it from pytestconfig is common enough that I want to cover it here.

We’ll use the pytest hook pytest_addoption to add a couple of options to the options already available in the pytest command line:

ch4/pytestconfig/conftest.py

```
def pytest_addoption(parser):
parser.addoption("--myopt", action="store_true",
help="some boolean option")
parser.addoption("--foo", action="store", default="bar",
help="foo: bar or baz")
```

Adding command-line options via pytest_addoption should be done via plugins or in the conftest.py file at the top of your project directory structure. You shouldn’t do it in a test subdirectory.

The options --myopt and --foo <value> were added to the previous code, and the help string was modified, as shown here:

```
$ cd /path/to/code/ch4/pytestconfig
$ pytest --help
usage: pytest [options] [file_or_dir] [file_or_dir] [...]
...
custom options:
--myopt some boolean option
--foo=FOO foo: bar or baz
...
```

Now we can access those options from a test:

ch4/pytestconfig/test_config.py

```
import pytest
```

```
def test_option(pytestconfig):
print('"foo" set to:', pytestconfig.getoption('foo'))
print('"myopt" set to:', pytestconfig.getoption('myopt'))
```

Let’s see how this works:

```
$ pytest -s -q test_config.py::test_option
"foo" set to: bar
"myopt" set to: False
.
1 passed in 0.01 seconds
$ pytest -s -q --myopt test_config.py::test_option
"foo" set to: bar
"myopt" set to: True
.
1 passed in 0.01 seconds
$ pytest -s -q --myopt --foo baz test_config.py::test_option
"foo" set to: baz
"myopt" set to: True
.
1 passed in 0.01 seconds
```

Because pytestconfig is a fixture, it can also be accessed from other fixtures. You can make fixtures for the option names, if you like, like this:

ch4/pytestconfig/test_config.py

```
@pytest.fixture()
def foo(pytestconfig):
return pytestconfig.option.foo


@pytest.fixture()
def myopt(pytestconfig):
return pytestconfig.option.myopt


def test_fixtures_for_options(foo, myopt):
print('"foo" set to:', foo)
print('"myopt" set to:', myopt)
```

You can also access builtin options, not just options you add, as well as information about how pytest was started (the directory, the arguments, and so on).

Here’s an example of a few configuration values and options:

ch4/pytestconfig/test_config.py

```
def test_pytestconfig(pytestconfig):
print('args :', pytestconfig.args)
print('inifile :', pytestconfig.inifile)
print('invocation_dir :', pytestconfig.invocation_dir)
print('rootdir :', pytestconfig.rootdir)
print('-k EXPRESSION :', pytestconfig.getoption('keyword'))
print('-v, --verbose :', pytestconfig.getoption('verbose'))
print('-q, --quiet :', pytestconfig.getoption('quiet'))
print('-l, --showlocals:', pytestconfig.getoption('showlocals'))
print('--tb=style :', pytestconfig.getoption('tbstyle'))
```

You’ll use pytestconfig again when I demonstrate ini files in Chapter 6, *Configuration*.

Using cache

Usually we testers like to think about each test as being as independent as possible from other tests. We want to make sure order dependencies don't creep in. We want to be able to run or rerun any test in any order and get the same result. We also want test sessions to be repeatable and to not change behavior based on previous test sessions.

However, sometimes passing information from one test session to the next can be quite useful. When we do want to pass information to future test sessions, we can do it with the cache builtin fixture.

The cache fixture is all about storing information about one test session and retrieving it in the next. A great example of using the powers of cache for good is the builtin functionality of --last-failed and --failed-first. Let's take a look at how the data for these flags is stored using cache.

Here's the help text for the --last-failed and --failed-first options, as well as a couple of cache options:

```
$ pytest --help
...
--lf, --last-failed rerun only the tests that failed at the last run (or
all if none failed)
--ff, --failed-first run all tests but run the last failures first. This
may re-order tests and thus lead to repeated fixture
setup/teardown
--cache-show show cache contents, don't perform collection or tests
--cache-clear remove all cache contents at start of test run.
...
```

To see these in action, we'll use these two tests:

ch4/cache/test_pass_fail.py

```
def test_this_passes():
assert 1 == 1


def test_this_fails():
assert 1 == 2
```

Let's run them using --verbose to see the function names, and --tb=no to hide the stack trace:

```
$ cd /path/to/code/ch4/cache
$ pytest --verbose --tb=no test_pass_fail.py
==================== test session starts ====================
collected 2 items
```

```
test_pass_fail.py::test_this_passes PASSED
test_pass_fail.py::test_this_fails FAILED

============ 1 failed, 1 passed in 0.05 seconds =============
```

If you run them again with the --ff or --failed-first flag, the tests that failed previously will be run first, followed by the rest of the session:

```
$ pytest --verbose --tb=no --ff test_pass_fail.py
==================== test session starts ====================
run-last-failure: rerun last 1 failures first
collected 2 items

test_pass_fail.py::test_this_fails FAILED
test_pass_fail.py::test_this_passes PASSED

============ 1 failed, 1 passed in 0.04 seconds =============
```

Or you can use --lf or --last-failed to just run the tests that failed the last time:

```
$ pytest --verbose --tb=no --lf test_pass_fail.py
==================== test session starts ====================
run-last-failure: rerun last 1 failures
collected 2 items

test_pass_fail.py::test_this_fails FAILED

==================== 1 tests deselected =====================
========== 1 failed, 1 deselected in 0.05 seconds ===========
```

Before we look at how the failure data is being saved and how you can use the same mechanism, let’s look at another example that makes the value of --lf and --ff even more obvious.

Here’s a parametrized test with one failure:

ch4/cache/test_few_failures.py

```
"""Demonstrate -lf and -ff with failing tests."""

import pytest
from pytest import approx


testdata = [
# x, y, expected
```

```
(1.01, 2.01, 3.02),
(1e25, 1e23, 1.1e25),
(1.23, 3.21, 4.44),
(0.1, 0.2, 0.3),
(1e25, 1e24, 1.1e25)
]


@pytest.mark.parametrize("x,y,expected", testdata)
def test_a(x, y, expected):
"""Demo approx()."""
sum_ = x + y
assert sum_ == approx(expected)
```

And the output:

```
$ cd /path/to/code/ch4/cache
$ pytest -q test_few_failures.py
.F...
====================== FAILURES ======================
_____________ test_a[1e+25-1e+23-1.1e+25] _____________

x = 1e+25, y = 1e+23, expected = 1.1e+25

@pytest.mark.parametrize("x,y,expected", testdata)
def test_a(x,y,expected):
sum_ = x + y
> assert sum_ == approx(expected)
E assert 1.01e+25 == 1.1e+25 ± 1.1e+19
E + where 1.1e+25 ± 1.1e+19 = approx(1.1e+25)

test_few_failures.py:17: AssertionError
1 failed, 4 passed in 0.06 seconds
```

Maybe you can spot the problem right off the bat. But let's pretend the test is longer and more complicated, and it's not obvious what's wrong. Let's run the test again to see the failure again. You can specify the test case on the command line:

```
$ pytest -q "test_few_failures.py::test_a[1e+25-1e+23-1.1e+25]"
```

If you don't want to copy/paste or there are multiple failed cases you'd like to rerun, --lf is much easier. And if you're really debugging a test failure, another flag that might make things easier is --showlocals, or -l for short:

```
$ pytest -q --lf -l test_few_failures.py
F
====================== FAILURES ======================
_____________ test_a[1e+25-1e+23-1.1e+25] _____________

x = 1e+25, y = 1e+23, expected = 1.1e+25

@pytest.mark.parametrize("x,y,expected", testdata)
def test_a(x,y,expected):
sum_ = x + y
> assert sum_ == approx(expected)
E assert 1.01e+25 == 1.1e+25 ± 1.1e+19
E + where 1.1e+25 ± 1.1e+19 = approx(1.1e+25)

expected = 1.1e+25
sum_ = 1.01e+25
x = 1e+25
y = 1e+23

test_few_failures.py:17: AssertionError
================= 4 tests deselected =================
1 failed, 4 deselected in 0.05 seconds
```

The reason for the failure should be more obvious now.

To pull off the trick of remembering what test failed last time, pytest stores test failure information from the last test session. You can see the stored information with --cache-show:

```
$ pytest --cache-show
===================== test session starts =====================
------------------------ cache values ------------------------
cache/lastfailed contains:
{'test_few_failures.py::test_a[1e+25-1e+23-1.1e+25]': True}

================= no tests ran in 0.00 seconds =================
```

Or you can look in the cache dir:

```
$ cat .cache/v/cache/lastfailed
{
"test_few_failures.py::test_a[1e+25-1e+23-1.1e+25]": true
}
```

You can pass in --clear-cache to clear the cache before the session.

The cache can be used for more than just --lf and --ff. Let's make a fixture that records how long tests take, saves the times, and on the next run, reports an error on tests that take longer than, say, twice as long as last time.

The interface for the cache fixture is simply

```
cache.get(key, default)
cache.set(key, value)
```

By convention, key names start with the name of your application or plugin, followed by a /, and continuing to separate sections of the key name with /'s. The value you store can be anything that is convertible to json, since that's how it's represented in the .cache directory.

Here's our fixture used to time tests:

ch4/cache/test_slower.py

```
@pytest.fixture(autouse=True)
def check_duration(request, cache):
key = 'duration/' + request.node.nodeid.replace(':', '_')
# nodeid's can have colons
# keys become filenames within .cache
# replace colons with something filename safe
start_time = datetime.datetime.now()
yield
stop_time = datetime.datetime.now()
this_duration = (stop_time - start_time).total_seconds()
last_duration = cache.get(key, None)
cache.set(key, this_duration)
if last_duration is not None:
errorstring = "test duration over 2x last duration"
assert this_duration <= last_duration * 2, errorstring
```

The fixture is autouse, so it doesn't need to be referenced from the test. The request object is used to grab the nodeid for use in the key. The nodeid is a unique identifier that works even with parametrized tests. We prepend the key with 'duration/' to be good cache citizens. The code above yield runs before the test function; the code after yield happens after the test function.

Now we need some tests that take different amounts of time:

ch4/cache/test_slower.py

```
@pytest.mark.parametrize('i', range(5))
def test_slow_stuff(i):
time.sleep(random.random())
```

Because you probably don't want to write a bunch of tests for this, I used random and parametrization to easily generate some tests that sleep for a random amount of time, all shorter than a second. Let's see it run a couple of times:

```
$ cd /path/to/code/ch4/cache
$ pytest -q --cache-clear test_slower.py
.....
5 passed in 2.10 seconds
$ pytest -q --tb=line test_slower.py
.E..E.E.



============================ ERRORS ============================
_____________ ERROR at teardown of test_slow_stuff[0] _____________
E AssertionError: test duration over 2x last duration
assert 0.954312 <= (0.380536 * 2)
_____________ ERROR at teardown of test_slow_stuff[2] _____________
E AssertionError: test duration over 2x last duration
assert 0.821745 <= (0.152405 * 2)
_____________ ERROR at teardown of test_slow_stuff[3] _____________
E AssertionError: test duration over 2x last duration
assert 1.001032 <= (0.36674 * 2)
5 passed, 3 error in 3.83 seconds
```

Well, that was fun. Let's see what's in the cache:

```
$ pytest -q --cache-show
------------------------ cache values -------------------------
cache/lastfailed contains:
{'test_slower.py::test_slow_stuff[0]': True,
'test_slower.py::test_slow_stuff[2]': True,
'test_slower.py::test_slow_stuff[3]': True}
duration/test_slower.py__test_slow_stuff[0] contains:
0.954312
duration/test_slower.py__test_slow_stuff[1] contains:
0.915539
duration/test_slower.py__test_slow_stuff[2] contains:
0.821745
duration/test_slower.py__test_slow_stuff[3] contains:
1.001032
duration/test_slower.py__test_slow_stuff[4] contains:
0.031884

no tests ran in 0.01 seconds
```

You can easily see the duration data separate from the cache data due to the prefixing of cache

data names. However, it's interesting that the lastfailed functionality is able to operate with one cache entry. Our duration data is taking up one cache entry per test. Let's follow the lead of lastfailed and fit our data into one entry.

We are reading and writing to the cache for every test. We could split up the fixture into a function scope fixture to measure durations and a session scope fixture to read and write to the cache. However, if we do this, we can't use the cache fixture because it has function scope. Fortunately, a quick peek at the implementation on GitHub[9] reveals that the cache fixture is simply returning request.config.cache. This is available in any scope.

Here's one possible refactoring of the same functionality:

ch4/cache/test_slower_2.py

```
Duration = namedtuple('Duration', ['current', 'last'])


@pytest.fixture(scope='session')
def duration_cache(request):
key = 'duration/testdurations'
d = Duration({}, request.config.cache.get(key, {}))
yield d
request.config.cache.set(key, d.current)


@pytest.fixture(autouse=True)
def check_duration(request, duration_cache):
d = duration_cache
nodeid = request.node.nodeid
start_time = datetime.datetime.now()
yield
duration = (datetime.datetime.now() - start_time).total_seconds()
d.current[nodeid] = duration
if d.last.get(nodeid, None) is not None:
errorstring = "test duration over 2x last duration"
assert duration <= (d.last[nodeid] * 2), errorstring
```

The duration_cache fixture is session scope. It reads the previous entry or an empty dictionary if there is no previous cached data, before any tests are run. In the previous code, we saved both the retrieved dictionary and an empty one in a namedtuple called Duration with accessors current and last. We then passed that namedtuple to the check_duration fixture, which is function scope and runs for every test function. As the test runs, the same namedtuple is passed to each test, and the times for the current test runs are stored in the d.current dictionary. At the end of the test session, the collected current dictionary is saved in the cache.

After running it a couple of times, let's look at the saved cache:

```
$ pytest -q --cache-clear test_slower_2.py
.....
5 passed in 2.80 seconds
$ pytest -q --tb=no test_slower_2.py
...E..E
5 passed, 2 error in 1.97 seconds
$ pytest -q --cache-show
------------------------ cache values -------------------------
cache/lastfailed contains:
{'test_slower_2.py::test_slow_stuff[2]': True,
'test_slower_2.py::test_slow_stuff[4]': True}
duration/testdurations contains:
{'test_slower_2.py::test_slow_stuff[0]': 0.145404,
'test_slower_2.py::test_slow_stuff[1]': 0.199585,
'test_slower_2.py::test_slow_stuff[2]': 0.696492,
'test_slower_2.py::test_slow_stuff[3]': 0.202118,
'test_slower_2.py::test_slow_stuff[4]': 0.657917}

no tests ran in 0.01 seconds
```

That looks better.

Using capsys

The capsys builtin fixture provides two bits of functionality: it allows you to retrieve stdout and stderr from some code, and it disables output capture temporarily. Let's take a look at retrieving stdout and stderr.

Suppose you have a function to print a greeting to stdout:

ch4/cap/test_capsys.py

```
def greeting(name):
print('Hi, {}'.format(name))
```

You can't test it by checking the return value. You have to test stdout somehow. You can test the output by using capsys:

ch4/cap/test_capsys.py

```
def test_greeting(capsys):
greeting('Earthling')
out, err = capsys.readouterr()
assert out == 'Hi, Earthling\n'
assert err == ''

greeting('Brian')
greeting('Nerd')
out, err = capsys.readouterr()
assert out == 'Hi, Brian\nHi, Nerd\n'
assert err == ''
```

The captured stdout and stderr are retrieved from capsys.redouterr(). The return value is whatever has been captured since the beginning of the function, or from the last time it was called.

The previous example only used stdout. Let's look at an example using stderr:

ch4/cap/test_capsys.py

```
def yikes(problem):
print('YIKES! {}'.format(problem), file=sys.stderr)


def test_yikes(capsys):
yikes('Out of coffee!')
out, err = capsys.readouterr()
assert out == ''
assert 'Out of coffee!' in err
```

pytest usually captures the output from your tests and the code under test. This includes print statements. The captured output is displayed for failing tests only after the full test session is complete. The -s option turns off this feature, and output is sent to stdout while the tests are running. Usually this works great, as it's the output from the failed tests you need to see in order to debug the failures. However, you may want to allow some output to make it through the default pytest output capture, to print some things without printing everything. You can do this with capsys. You can use capsys.disabled() to temporarily let output get past the capture mechanism.

Here's an example:

ch4/cap/test_capsys.py

```
def test_capsys_disabled(capsys):
with capsys.disabled():
print('\nalways print this')
print('normal print, usually captured')
```

Now, 'always print this' will always be output:

```
$ cd /path/to/code/ch4/cap
$ pytest -q test_capsys.py::test_capsys_disabled

always print this
.
1 passed in 0.01 seconds
$ pytest -q -s test_capsys.py::test_capsys_disabled

always print this
normal print, usually captured
.
1 passed in 0.00 seconds
```

As you can see, always print this shows up with or without output capturing, since it's being printed from within a with capsys.disabled() block. The other print statement is just a normal print statement, so normal print, usually captured is only seen in the output when we pass in the -s flag, which is a shortcut for --capture=no, turning off output capture.

Using monkeypatch

A “monkey patch” is a dynamic modification of a class or module during runtime. During testing, “monkey patching” is a convenient way to take over part of the runtime environment of the code under test and replace either input dependencies or output dependencies with objects or functions that are more convenient for testing. The monkeypatch builtin fixture allows you to do this in the context of a single test. And when the test ends, regardless of pass or fail, the original unpatched is restored, undoing everything changed by the patch. It’s all very hand-wavy until we jump into some examples. After looking at the API, we’ll look at how monkeypatch is used in test code.

The monkeypatch fixture provides the following functions:

- setattr(target, name, value=<notset>, raising=True): Set an attribute.
- delattr(target, name=<notset>, raising=True): Delete an attribute.
- setitem(dic, name, value): Set a dictionary entry.
- delitem(dic, name, raising=True): Delete a dictionary entry.
- setenv(name, value, prepend=None): Set an environmental variable.
- delenv(name, raising=True): Delete an environmental variable.
- syspath_prepend(path): Prepend path to sys.path, which is Python’s list of import locations.
- chdir(path): Change the current working directory.

The raising parameter tells pytest whether or not to raise an exception if the item doesn’t already exist. The prepend parameter to setenv() can be a character. If it is set, the value of the environmental variable will be changed to value + prepend + <old value>.

To see monkeypatch in action, let’s look at code that writes a dot configuration file. The behavior of some programs can be changed with preferences and values set in a dot file in a user’s home directory. Here’s a bit of code that reads and writes a cheese preferences file:

ch4/monkey/cheese.py

```
import os
import json


def read_cheese_preferences():
full_path = os.path.expanduser('~/.cheese.json')
with open(full_path, 'r') as f:
prefs = json.load(f)
return prefs
```

```
  def write_cheese_preferences(prefs):
  full_path = os.path.expanduser('~/.cheese.json')
  with open(full_path, 'w') as f:
  json.dump(prefs, f, indent=4)


  def write_default_cheese_preferences():
  write_cheese_preferences(_default_prefs)
  _default_prefs = {
  'slicing': ['manchego', 'sharp cheddar'],
  'spreadable': ['Saint Andre', 'camembert',
  'bucheron', 'goat', 'humbolt fog', 'cambozola'],
  'salads': ['crumbled feta']
  }
```

Let's take a look at how we could test write_default_cheese_preferences(). It's a function that takes no parameters and doesn't return anything. But it does have a side effect that we can test. It writes a file to the current user's home directory.

One approach is to just let it run normally and check the side effect. Suppose I already have tests for read_cheese_preferences() and I trust them, so I can use them in the testing of write_default_cheese_preferences():

ch4/monkey/test_cheese.py

```
  def test_def_prefs_full():
  cheese.write_default_cheese_preferences()
  expected = cheese._default_prefs
  actual = cheese.read_cheese_preferences()
  assert expected == actual
```

One problem with this is that anyone who runs this test code will overwrite their own cheese preferences file. That's not good.

If a user has HOME set, os.path.expanduser() replaces ~ with whatever is in a user's HOME environmental variable. Let's create a temporary directory and redirect HOME to point to that new temporary directory:

ch4/monkey/test_cheese.py

```
  def test_def_prefs_change_home(tmpdir, monkeypatch):
  monkeypatch.setenv('HOME', tmpdir.mkdir('home'))
  cheese.write_default_cheese_preferences()
  expected = cheese._default_prefs
  actual = cheese.read_cheese_preferences()
```

```
  assert expected == actual
```

This is a pretty good test, but relying on HOME seems a little operating-system dependent. And a peek into the documentation online for expanduser() has some troubling information, including "On Windows, HOME and USERPROFILE will be used if set, otherwise a combination of…."[10] Dang. That may not be good for someone running the test on Windows. Maybe we should take a different approach.

Instead of patching the HOME environmental variable, let's patch expanduser:

ch4/monkey/test_cheese.py

```
  def test_def_prefs_change_expanduser(tmpdir, monkeypatch):
  fake_home_dir = tmpdir.mkdir('home')
  monkeypatch.setattr(cheese.os.path, 'expanduser',
  (lambda x: x.replace('~', str(fake_home_dir))))
  cheese.write_default_cheese_preferences()
  expected = cheese._default_prefs
  actual = cheese.read_cheese_preferences()
  assert expected == actual
```

During the test, anything in the cheese module that calls os.path.expanduser() gets our lambda expression instead. This little function uses the regular expression module function re.sub to replace ~ with our new temporary directory. Now we've used setenv() and setattr() to do patching of environmental variables and attributes. Next up, setitem().

Let's say we're worried about what happens if the file already exists. We want to be sure it gets overwritten with the defaults when write_default_cheese_preferences() is called:

ch4/monkey/test_cheese.py

```
  def test_def_prefs_change_defaults(tmpdir, monkeypatch):
  # write the file once
  fake_home_dir = tmpdir.mkdir('home')
  monkeypatch.setattr(cheese.os.path, 'expanduser',
  (lambda x: x.replace('~', str(fake_home_dir))))
  cheese.write_default_cheese_preferences()
  defaults_before = copy.deepcopy(cheese._default_prefs)

  # change the defaults
  monkeypatch.setitem(cheese._default_prefs, 'slicing', ['provolone'])
  monkeypatch.setitem(cheese._default_prefs, 'spreadable', ['brie'])
  monkeypatch.setitem(cheese._default_prefs, 'salads', ['pepper jack'])
  defaults_modified = cheese._default_prefs

  # write it again with modified defaults
  cheese.write_default_cheese_preferences()
```

```
    # read, and check
    actual = cheese.read_cheese_preferences()
    assert defaults_modified == actual
    assert defaults_modified != defaults_before
```

Because _default_prefs is a dictionary, we can use monkeypatch.setitem() to change dictionary items just for the duration of the test.

We've used setenv(), setattr(), and setitem(). The del forms are pretty similar. They just delete an environmental variable, attribute, or dictionary item instead of setting something. The last two monkeypatch methods pertain to paths.

syspath_prepend(path) prepends a path to sys.path, which has the effect of putting your new path at the head of the line for module import directories. One use for this would be to replace a system-wide module or package with a stub version. You can then use monkeypatch.syspath_prepend() to prepend the directory of your stub version and the code under test will find the stub version first.

chdir(path) changes the current working directory during the test. This would be useful for testing command-line scripts and other utilities that depend on what the current working directory is. You could set up a temporary directory with whatever contents make sense for your script, and then use monkeypatch.chdir(the_tmpdir).

You can also use the monkeypatch fixture functions in conjunction with unittest.mock to temporarily replace attributes with mock objects. You'll look at that in Chapter 7, *Using pytest with Other Tools*.

Using doctest_namespace

The doctest module is part of the standard Python library and allows you to put little code examples inside docstrings for a function and test them to make sure they work. You can have pytest look for and run doctest tests within your Python code by using the --doctest-modules flag. With the doctest_namespace builtin fixture, you can build autouse fixtures to add symbols to the namespace pytest uses while running doctest tests. This allows docstrings to be much more readable. doctest_namespace is commonly used to add module imports into the namespace, especially when Python convention is to shorten the module or package name. For instance, numpy is often imported with import numpy as np.

Let’s play with an example. Let’s say we have a module named unnecessary_math.py with multiply() and divide() methods that we really want to make sure everyone understands clearly. So we throw some usage examples in both the file docstring and the docstrings of the functions:

ch4/dt/1/unnecessary_math.py

```
"""
This module defines multiply(a, b) and divide(a, b).

>>> import unnecessary_math as um

Here's how you use multiply:

>>> um.multiply(4, 3)
12
>>> um.multiply('a', 3)
'aaa'


Here's how you use divide:

>>> um.divide(10, 5)
2.0
"""


def multiply(a, b):
"""
 Returns a multiplied by b.

 >>> um.multiply(4, 3)
 12
 >>> um.multiply('a', 3)
```

```
    'aaa'
    """
    return a * b


def divide(a, b):
    """
    Returns a divided by b.

    >>> um.divide(10, 5)
    2.0
    """
    return a / b
```

Since the name unnecessary_math is long, we decide to use um instead by using import unnecessary_math as um in the top docstring. The code in the docstrings of the functions doesn't include the import statement, but continue with the um convention. The problem is that pytest treats each docstring with code as a different test. The import in the top docstring will allow the first part to pass, but the code in the docstrings of the functions will fail:

```
$ cd /path/to/code/ch4/dt/1
$ pytest -v --doctest-modules --tb=short unnecessary_math.py
======================== test session starts ========================
collected 3 items

unnecessary_math.py::unnecessary_math PASSED
unnecessary_math.py::unnecessary_math.divide FAILED
unnecessary_math.py::unnecessary_math.multiply FAILED

============================== FAILURES
==============================
_____________________ [doctest] unnecessary_math.divide _____________________
031
032 Returns a divided by b.
033
034 >>> um.divide(10, 5)
UNEXPECTED EXCEPTION: NameError("name 'um' is not defined",)
Traceback (most recent call last):
...
File "<doctest unnecessary_math.divide[0]>", line 1, in <module>

NameError: name 'um' is not defined
```

```
/path/to/code/ch4/dt/1/unnecessary_math.py:34: UnexpectedException
_____ [doctest] unnecessary_math.multiply _____
022
023 >>> um.multiply(4, 3)
UNEXPECTED EXCEPTION: NameError("name 'um' is not defined",)
Traceback (most recent call last):
...
File "<doctest unnecessary_math.multiply[0]>", line 1, in <module>

NameError: name 'um' is not defined

/path/to/code/ch4/dt/1/unnecessary_math.py:23: UnexpectedException
================ 2 failed, 1 passed in 0.03 seconds =================
```

One way to fix it is to put the import statement in each docstring:

ch4/dt/2/unnecessary_math.py

```
def multiply(a, b):
"""
 Returns a multiplied by b.

 >>> import unnecessary_math as um
 >>> um.multiply(4, 3)
 12
 >>> um.multiply('a', 3)
 'aaa'
 """
return a * b


def divide(a, b):
"""
 Returns a divided by b.

 >>> import unnecessary_math as um
 >>> um.divide(10, 5)
 2.0
 """
return a / b
```

This definitely fixes the problem:

```
$ cd /path/to/code/ch4/dt/2
```

```
$ pytest -v --doctest-modules --tb=short unnecessary_math.py
======================== test session starts ========================
collected 3 items

unnecessary_math.py::unnecessary_math PASSED
unnecessary_math.py::unnecessary_math.divide PASSED
unnecessary_math.py::unnecessary_math.multiply PASSED

===================== 3 passed in 0.03 seconds ======================
```

However, it also clutters the docstrings, and doesn't add any real value to readers of the code.

The builtin fixture doctest_namespace, used in an autouse fixture at a top-level conftest.py file, will fix the problem without changing the source code:

ch4/dt/3/conftest.py

```
import pytest
import unnecessary_math


@pytest.fixture(autouse=True)
def add_um(doctest_namespace):
doctest_namespace['um'] = unnecessary_math
```

This tells pytest to add the um name to the doctest_namespace and have it be the value of the imported unnecessary_math module. With this in place in the conftest.py file, any doctests found within the scope of this conftest.py file will have the um symbol defined.

I'll cover running doctest from pytest more in Chapter 7, *Using pytest with Other Tools*.

Using recwarn

The recwarn builtin fixture is used to examine warnings generated by code under test. In Python, you can add warnings that work a lot like assertions, but are used for things that don't need to stop execution. For example, suppose we want to stop supporting a function that we wish we had never put into a package but was released for others to use. We can put a warning in the code and leave it there for a release or two:

ch4/test_warnings.py

```
import warnings
import pytest


def lame_function():
warnings.warn("Please stop using this", DeprecationWarning)
# rest of function
```

We can make sure the warning is getting issued correctly with a test:

ch4/test_warnings.py

```
def test_lame_function(recwarn):
lame_function()
assert len(recwarn) == 1
w = recwarn.pop()
assert w.category == DeprecationWarning
assert str(w.message) == 'Please stop using this'
```

The recwarn value acts like a list of warnings, and each warning in the list has a category, message, filename, and lineno defined, as shown in the code.

The warnings are collected at the beginning of the test. If that is inconvenient because the portion of the test where you care about warnings is near the end, you can use recwarn.clear() to clear out the list before the chunk of the test where you do care about collecting warnings.

In addition to recwarn, pytest can check for warnings with pytest.warns():

ch4/test_warnings.py

```
def test_lame_function_2():
with pytest.warns(None) as warning_list:
lame_function()


assert len(warning_list) == 1
w = warning_list.pop()
assert w.category == DeprecationWarning
assert str(w.message) == 'Please stop using this'
```

The pytest.warns() context manager provides an elegant way to demark what portion of the code you're checking warnings. The recwarn fixture and the pytest.warns() context manager provide similar functionality, though, so the decision of which to use is purely a matter of taste.

Exercises

1. In ch4/cache/test_slower.py, there is an autouse fixture called check_duration(). Copy it into ch3/tasks_proj/tests/conftest.py.
2. Run the tests in Chapter 3.
3. For tests that are really fast, 2x really fast is still really fast. Instead of 2x, change the fixture to check for 0.1 second plus 2x the last duration.
4. Run pytest with the modified fixture. Do the results seem reasonable?

What’s Next

In this chapter, you looked at many of pytest’s builtin fixtures. Next, you’ll take a closer look at plugins. The nuance of writing large plugins could be a book in itself; however, small custom plugins are a regular part of the pytest ecosystem.

Footnotes

[8] http://py.readthedocs.io/en/latest/path.html

[9] https://github.com/pytest-dev/pytest/blob/master/_pytest/cacheprovider.py

[10] https://docs.python.org/3.6/library/os.path.html#os.path.expanduser

Chapter 5
Plugins

As powerful as pytest is right out of the box, it gets even better when you add plugins to the mix. The pytest code base is structured with customization and extensions, and there are hooks available to allow modifications and improvements through plugins.

It might surprise you to know that you've already written some plugins if you've worked through the previous chapters in this book. Any time you put fixtures and/or hook functions into a project's top-level conftest.py file, you created a local conftest plugin. It's just a little bit of extra work to convert these conftest.py files into installable plugins that you can share between projects, with other people, or with the world.

We will start this chapter looking at where to look for third-party plugins. Quite a few plugins are available, so there's a decent chance someone has already written the change you want to make to pytest. Since we will be looking at open source plugins, if a plugin does almost what you want to do but not quite, you can fork it, or use it as a reference for creating your own plugin. While this chapter is about creating your own plugins, Appendix 3, *Plugin Sampler Pack* is included to give you a taste of what's possible.

In this chapter, you'll learn how to create plugins, and I'll point you in the right direction to test, package, and distribute them. The full topic of Python packaging and distribution is probably a book of its own, so we won't cover everything. But you'll get far enough to be able to share plugins with your team. I'll also discuss some shortcuts to getting PyPI–distributed plugins up with the least amount of work.

Finding Plugins

You can find third-party pytest plugins in several places. The plugins listed in Appendix 3, *Plugin Sampler Pack*, are all available for download from PyPI. However, that's not the only place to look for great pytest plugins.

https://docs.pytest.org/en/latest/plugins.html

> The main pytest documentation site has a page that talks about installing and using pytest plugins, and lists a few common plugins.

https://pypi.python.org

> The Python Package Index (PyPI) is a great place to get lots of Python packages, but it is also a great place to find pytest plugins. When looking for pytest plugins, it should work pretty well to enter "pytest," "pytest-," or "-pytest" into the search box, since most pytest plugins either start with "pytest-," or end in "-pytest."

https://github.com/pytest-dev

> The "pytest-dev" group on GitHub is where the pytest source code is kept. It's also where you can find some popular pytest plugins that are intended to be maintained long-term by the pytest core team.

Installing Plugins

pytest plugins are installed with pip, just like other Python packages. However, you can use pip in several different ways to install plugins.

Install from PyPI

As PyPI is the default location for pip, installing plugins from PyPI is the easiest method. Let's install the pytest-cov plugin:

```
$ pip install pytest-cov
```

This installs the latest stable version from PyPI.

Install a Particular Version from PyPI

If you want a particular version of a plugin, you can specify the version after '==':

```
$ pip install pytest-cov==2.4.0
```

Install from a .tar.gz or .whl File

Packages on PyPI are distributed as zipped files with the extensions .tar.gz and/or .whl. These are often referred to as "tar balls" and "wheels." If you're having trouble getting pip to work with PyPI directly (which can happen with firewalls and other network complications), you can download either the .tar.gz or the .whl and install from that.

You don't have to unzip or anything; just point pip at it:

```
$ pip install pytest-cov-2.4.0.tar.gz
# or
$ pip install pytest_cov-2.4.0-py2.py3-none-any.whl
```

Install from a Local Directory

You can keep a local stash of plugins (and other Python packages) in a local or shared directory in .tar.gz or .whl format and use that instead of PyPI for installing plugins:

```
$ mkdir some_plugins
$ cp pytest_cov-2.4.0-py2.py3-none-any.whl some_plugins/
$ pip install --no-index --find-links=./some_plugins/ pytest-cov
```

The --no-index tells pip to not connect to PyPI. The --find-links=./some_plugins/ tells pip to look in the directory called some_plugins. This technique is especially useful if you have both third-

party and your own custom plugins stored locally, and also if you're creating new virtual environments for continuous integration or with tox. (We'll talk about both tox and continuous integration in Chapter 7, *Using pytest with Other Tools*.)

Note that with the local directory install method, you can install multiple versions and specify which version you want by adding == and the version number:

```
$ pip install --no-index --find-links=./some_plugins/ pytest-cov==2.4.0
```

Install from a Git Repository

You can install plugins directly from a Git repository—in this case, GitHub:

```
$ pip install git+https://github.com/pytest-dev/pytest-cov
```

You can also specify a version tag:

```
$ pip install git+https://github.com/pytest-dev/pytest-cov@v2.4.0
```

Or you can specify a branch:

```
$ pip install git+https://github.com/pytest-dev/pytest-cov@master
```

Installing from a Git repository is especially useful if you're storing your own work within Git, or if the plugin or plugin version you want isn't on PyPI.

Writing Your Own Plugins

Many third-party plugins contain quite a bit of code. That's one of the reasons we use them—to save us the time to develop all of that code ourselves. However, for your specific coding domain, you'll undoubtedly come up with special fixtures and modifications that help you test. Even a handful of fixtures that you want to share between a couple of projects can be shared easily by creating a plugin. You can share those changes with multiple projects—and possibly the rest of the world—by developing and distributing your own plugins. It's pretty easy to do so. In this section, we'll develop a small modification to pytest behavior, package it as a plugin, test it, and look into how to distribute it.

Plugins can include hook functions that alter pytest's behavior. Because pytest was developed with the intent to allow plugins to change quite a bit about the way pytest behaves, a lot of hook functions are available. The hook functions for pytest are specified on the pytest documentation site.[11]

For our example, we'll create a plugin that changes the way the test status looks. We'll also include a command-line option to turn on this new behavior. We're also going to add some text to the output header. Specifically, we'll change all of the FAILED status indicators to "OPPORTUNITY for improvement," change F to O, and add "Thanks for running the tests" to the header. We'll use the --nice option to turn the behavior on.

To keep the behavior changes separate from the discussion of plugin mechanics, we'll make our changes in conftest.py before turning it into a distributable plugin. You don't have to start plugins this way. But frequently, changes you only intended to use on one project will become useful enough to share and grow into a plugin. Therefore, we'll start by adding functionality to a conftest.py file, then, after we get things working in conftest.py, we'll move the code to a package.

Let's go back to the Tasks project. In *Expecting Exceptions*, we wrote some tests that made sure exceptions were raised if someone called an API function incorrectly. Looks like we missed at least a few possible error conditions.

Here are a couple more tests:

ch5/a/tasks_proj/tests/func/test_api_exceptions.py

```
import pytest
import tasks
from tasks import Task


@pytest.mark.usefixtures('tasks_db')
class TestAdd():
"""Tests related to tasks.add()."""

def test_missing_summary(self):
"""Should raise an exception if summary missing."""
```

```
    with pytest.raises(ValueError):
    tasks.add(Task(owner='bob'))

    def test_done_not_bool(self):
    """Should raise an exception if done is not a bool."""
    with pytest.raises(ValueError):
    tasks.add(Task(summary='summary', done='True'))
```

Let’s run them to see if they pass:

```
    $ cd /path/to/code/ch5/a/tasks_proj
    $ pytest
    ====================== test session starts ======================
    collected 57 items

    tests/func/test_add.py ...
    tests/func/test_add_variety.py ............................
    tests/func/test_add_variety2.py ............
    tests/func/test_api_exceptions.py .F.......
    tests/func/test_unique_id.py .
    tests/unit/test_task.py ....

    ============================ FAILURES ============================
    _____________________ TestAdd.test_done_not_bool _____________________

    self = <func.test_api_exceptions.TestAdd object at 0x103a71a20>

    def test_done_not_bool(self):
    """Should raise an exception if done is not a bool."""
    with pytest.raises(ValueError):
    > tasks.add(Task(summary='summary', done='True'))
    E Failed: DID NOT RAISE <class 'ValueError'>

    tests/func/test_api_exceptions.py:20: Failed
    ============== 1 failed, 56 passed in 0.28 seconds ==============
```

Let’s run it again with -v for verbose. Since you’ve already seen the traceback, you can turn that off with --tb=no.

And now let’s focus on the new tests with -k TestAdd, which works because there aren’t any other tests with names that contain “TestAdd.”

```
    $ cd /path/to/code/ch5/a/tasks_proj/tests/func
    $ pytest -v --tb=no test_api_exceptions.py -k TestAdd
```

```
===================== test session starts =====================
collected 9 items

test_api_exceptions.py::TestAdd::test_missing_summary PASSED
test_api_exceptions.py::TestAdd::test_done_not_bool FAILED

====================== 7 tests deselected ======================
======= 1 failed, 1 passed, 7 deselected in 0.07 seconds =======
```

We could go off and try to fix this test (and we should later), but now we are focused on trying to make failures more pleasant for developers.

Let's start by adding the "thank you" message to the header, which you can do with a pytest hook called pytest_report_header().

ch5/b/tasks_proj/tests/conftest.py

```
def pytest_report_header():
"""Thank tester for running tests."""
return "Thanks for running the tests."
```

Obviously, printing a thank-you message is rather silly. However, the ability to add information to the header can be extended to add a username and specify hardware used and versions under test. Really, anything you can convert to a string, you can stuff into the test header.

Next, we'll change the status reporting for tests to change F to O and FAILED to OPPORTUNITY for improvement. There's a hook function that allows for this type of shenanigans: pytest_report_teststatus():

ch5/b/tasks_proj/tests/conftest.py

```
def pytest_report_teststatus(report):
"""Turn failures into opportunities."""
if report.when == 'call' and report.failed:
return (report.outcome, 'O', 'OPPORTUNITY for improvement')
```

And now we have just the output we were looking for. A test session with no --verbose flag shows an O for failures, er, improvement opportunities:

```
$ cd /path/to/code/ch5/b/tasks_proj/tests/func
$ pytest --tb=no test_api_exceptions.py -k TestAdd
===================== test session starts =====================
Thanks for running the tests.
collected 9 items

test_api_exceptions.py .O

====================== 7 tests deselected ======================
```

```
======= 1 failed, 1 passed, 7 deselected in 0.06 seconds =======
```

And the -v or --verbose flag will be nicer also:

```
$ pytest -v --tb=no test_api_exceptions.py -k TestAdd
==================== test session starts ====================
Thanks for running the tests.
collected 9 items

test_api_exceptions.py::TestAdd::test_missing_summary PASSED
test_api_exceptions.py::TestAdd::test_done_not_bool OPPORTUNITY for improvement

====================== 7 tests deselected ======================
======= 1 failed, 1 passed, 7 deselected in 0.07 seconds =======
```

The last modification we'll make is to add a command-line option, --nice, to only have our status modifications occur if --nice is passed in:

ch5/c/tasks_proj/tests/conftest.py

```
def pytest_addoption(parser):
"""Turn nice features on with --nice option."""
group = parser.getgroup('nice')
group.addoption("--nice", action="store_true",
help="nice: turn failures into opportunities")


def pytest_report_header():
"""Thank tester for running tests."""
if pytest.config.getoption('nice'):
return "Thanks for running the tests."


def pytest_report_teststatus(report):
"""Turn failures into opportunities."""
if report.when == 'call':
if report.failed and pytest.config.getoption('nice'):
return (report.outcome, 'O', 'OPPORTUNITY for improvement')
```

This is a good place to note that for this plugin, we are using just a couple of hook functions. There are many more, which can be found on the main pytest documentation site.[12]

We can manually test our plugin just by running it against our example file. First, with no --nice option, to make sure just the username shows up:

```
$ cd /path/to/code/ch5/c/tasks_proj/tests/func
```

```
$ pytest --tb=no test_api_exceptions.py -k TestAdd
==================== test session starts ====================
collected 9 items

test_api_exceptions.py .F

====================== 7 tests deselected ======================
======= 1 failed, 1 passed, 7 deselected in 0.07 seconds =======
```

Now with --nice:

```
$ pytest --nice --tb=no test_api_exceptions.py -k TestAdd
==================== test session starts ====================
Thanks for running the tests.
collected 9 items

test_api_exceptions.py .O

====================== 7 tests deselected ======================
======= 1 failed, 1 passed, 7 deselected in 0.07 seconds =======
```

And with --nice and --verbose:

```
$ pytest -v --nice --tb=no test_api_exceptions.py -k TestAdd
==================== test session starts ====================
Thanks for running the tests.
collected 9 items

test_api_exceptions.py::TestAdd::test_missing_summary PASSED
test_api_exceptions.py::TestAdd::test_done_not_bool OPPORTUNITY for improvement

====================== 7 tests deselected ======================
======= 1 failed, 1 passed, 7 deselected in 0.06 seconds =======
```

Great! All of the changes we wanted are done with about a dozen lines of code in a conftest.py file. Next, we'll move this code into a plugin structure.

Creating an Installable Plugin

The process for sharing plugins with others is well-defined. Even if you never put your own plugin up on PyPI, by walking through the process, you'll have an easier time reading the code from open source plugins and be better equipped to judge if they will help you or not.

It would be overkill to fully cover Python packaging and distribution in this book, as the topic is well documented elsewhere.[13][14] However, it's a small task to go from the local config plugin we created in the previous section to something pip-installable.

First, we need to create a new directory to put our plugin code. It does not matter what you call it, but since we are making a plugin for the "nice" flag, let's call it pytest-nice. We will have two files in this new directory: pytest_nice.py and setup.py. (The tests directory will be discussed in *Testing Plugins*.)

```
pytest-nice
├── LICENCE
├── README.rst
├── pytest_nice.py
├── setup.py
└── tests
    ├── conftest.py
    └── test_nice.py
```

In pytest_nice.py, we'll put the exact contents of our conftest.py that were related to this feature (and take it out of the tasks_proj/tests/conftest.py):

ch5/pytest-nice/pytest_nice.py

```
"""Code for pytest-nice plugin."""

import pytest


def pytest_addoption(parser):
"""Turn nice features on with --nice option."""
group = parser.getgroup('nice')
group.addoption("--nice", action="store_true",
help="nice: turn FAILED into OPPORTUNITY for improvement")


def pytest_report_header():
"""Thank tester for running tests."""
if pytest.config.getoption('nice'):
return "Thanks for running the tests."
```

```
  def pytest_report_teststatus(report):
  """Turn failures into opportunities."""
  if report.when == 'call':
  if report.failed and pytest.config.getoption('nice'):
  return (report.outcome, 'O', 'OPPORTUNITY for improvement')
```

In setup.py, we need a very minimal call to setup():

ch5/pytest-nice/setup.py

```
  """Setup for pytest-nice plugin."""

  from setuptools import setup

  setup(
  name='pytest-nice',
  version='0.1.0',
  description='A pytest plugin to turn FAILURE into OPPORTUNITY',
  url='https://wherever/you/have/info/on/this/package',
  author='Your Name',
  author_email='your_email@somewhere.com',
  license='proprietary',
  py_modules=['pytest_nice'],
  install_requires=['pytest'],
  entry_points={'pytest11': ['nice = pytest_nice', ], },
  )
```

You'll want more information in your setup if you're going to distribute to a wide audience or online. However, for a small team or just for yourself, this will suffice.

You can include many more parameters to setup(); we only have the required fields. The version field is the version of this plugin. And it's up to you when you bump the version. The url field is required. You can leave it out, but you get a warning if you do. The author and author_email fields can be replaced with maintainer and maintainer_email, but one of those pairs needs to be there. The license field is a short text field. It can be one of the many open source licenses, your name or company, or whatever is appropriate for you. The py_modules entry lists pytest_nice as our one and only module for this plugin. Although it's a list and you could include more than one module, if I had more than one, I'd use packages instead and put all the modules inside a directory.

So far, all of the parameters to setup() are standard and used for all Python installers. The piece that is different for pytest plugins is the entry_points parameter. We have listed entry_points= {'pytest11': ['nice = pytest_nice',], },. The entry_points feature is standard for setuptools, but pytest11 is a special identifier that pytest looks for. With this line, we are telling pytest that nice is the name of our plugin, and pytest_nice is the name of the module where our plugin lives. If

we had used a package, our entry here would be:

```
entry_points={'pytest11': ['name_of_plugin = myproject.pluginmodule',], },
```

I haven't talked about the README.rst file yet. Some form of README is a requirement by setuptools. If you leave it out, you'll get this:

```
...
warning: sdist: standard file not found: should have one of README,
README.rst, README.txt
...
```

Keeping a README around as a standard way to include some information about a project is a good idea anyway. Here's what I've put in the file for pytest-nice:

ch5/pytest-nice/README.rst

```
pytest-nice : A pytest plugin
============================

Makes pytest output just a bit nicer during failures.

Features
--------

- Includes user name of person running tests in pytest output.
- Adds ``--nice`` option that:

- turns ``F`` to ``O``
- with ``-v``, turns ``FAILURE`` to ``OPPORTUNITY for improvement``

Installation
------------

Given that our pytest plugins are being saved in .tar.gz form in the
shared directory PATH, then install like this:

::

$ pip install PATH/pytest-nice-0.1.0.tar.gz
$ pip install --no-index --find-links PATH pytest-nice

Usage
-----
```

::

```
$ pytest --nice
```

There are lots of opinions about what should be in a README. This is a rather minimal version, but it works.

Testing Plugins

Plugins are code that needs to be tested just like any other code. However, testing a change to a testing tool is a little tricky. When we developed the plugin code in *Writing Your Own Plugins*, we tested it manually by using a sample test file, running pytest against it, and looking at the output to make sure it was right. We can do the same thing in an automated way using a plugin called pytester that ships with pytest but is disabled by default.

Our test directory for pytest-nice has two files: conftest.py and test_nice.py. To use pytester, we need to add just one line to conftest.py:

ch5/pytest-nice/tests/conftest.py

```
"""pytester is needed for testing plugins."""
pytest_plugins = 'pytester'
```

This turns on the pytester plugin. We will be using a fixture called testdir that becomes available when pytester is enabled.

Often, tests for plugins take on the form we've described in manual steps:

1. Make an example test file.
2. Run pytest with or without some options in the directory that contains our example file.
3. Examine the output.
4. Possibly check the result code—0 for all passing, 1 for some failing.

Let's look at one example:

ch5/pytest-nice/tests/test_nice.py

```
def test_pass_fail(testdir):

# create a temporary pytest test module
testdir.makepyfile("""
 def test_pass():
 assert 1 == 1

 def test_fail():
 assert 1 == 2
 """)

# run pytest
result = testdir.runpytest()

# fnmatch_lines does an assertion internally
result.stdout.fnmatch_lines([
'*.F', # . for Pass, F for Fail
```

```
    ])

    # make sure that that we get a '1' exit code for the testsuite
    assert result.ret == 1
```

The testdir fixture automatically creates a temporary directory for us to put test files. It has a method called makepyfile() that allows us to put in the contents of a test file. In this case, we are creating two tests: one that passes and one that fails.

We run pytest against the new test file with testdir.runpytest(). You can pass in options if you want. The return value can then be examined further, and is of type RunResult.[15]

Usually, I look at stdout and ret. For checking the output like we did manually, use fnmatch_lines, passing in a list of strings that we want to see in the output, and then making sure that ret is 0 for passing sessions and 1 for failing sessions. The strings passed into fnmatch_lines can include glob wildcards. We can use our example file for more tests. Instead of duplicating that code, let's make a fixture:

ch5/pytest-nice/tests/test_nice.py

```
    @pytest.fixture()
    def sample_test(testdir):
    testdir.makepyfile("""
     def test_pass():
     assert 1 == 1

     def test_fail():
     assert 1 == 2
     """)
    return testdir
```

Now, for the rest of the tests, we can use sample_test as a directory that already contains our sample test file. Here are the tests for the other option variants:

ch5/pytest-nice/tests/test_nice.py

```
    def test_with_nice(sample_test):
    result = sample_test.runpytest('--nice')
    result.stdout.fnmatch_lines(['*.O', ]) # . for Pass, O for Fail
    assert result.ret == 1


    def test_with_nice_verbose(sample_test):
    result = sample_test.runpytest('-v', '--nice')
    result.stdout.fnmatch_lines([
    '*::test_fail OPPORTUNITY for improvement',
    ])
    assert result.ret == 1
```

```
  def test_not_nice_verbose(sample_test):
  result = sample_test.runpytest('-v')
  result.stdout.fnmatch_lines(['*::test_fail FAILED'])
  assert result.ret == 1
```

Just a couple more tests to write. Let's make sure our thank-you message is in the header:

ch5/pytest-nice/tests/test_nice.py

```
  def test_header(sample_test):
  result = sample_test.runpytest('--nice')
  result.stdout.fnmatch_lines(['Thanks for running the tests.'])


  def test_header_not_nice(sample_test):
  result = sample_test.runpytest()
  thanks_message = 'Thanks for running the tests.'
  assert thanks_message not in result.stdout.str()
```

This could have been part of the other tests also, but I like to have it in a separate test so that one test checks one thing.

Finally, let's check the help text:

ch5/pytest-nice/tests/test_nice.py

```
  def test_help_message(testdir):
  result = testdir.runpytest('--help')

  # fnmatch_lines does an assertion internally
  result.stdout.fnmatch_lines([
  'nice:',
  '*--nice*nice: turn FAILED into OPPORTUNITY for improvement',
  ])
```

I think that's a pretty good check to make sure our plugin works.

To run the tests, let's start in our pytest-nice directory and make sure our plugin is installed. We do this either by installing the .zip.gz file or installing the current directory in editable mode:

```
  $ cd /path/to/code/ch5/pytest-nice/
  $ pip install .
  Processing /path/to/code/ch5/pytest-nice
  Requirement already satisfied: pytest in
  /path/to/venv/lib/python3.6/site-packages (from pytest-nice==0.1.0)
```

```
Requirement already satisfied: py>=1.4.33 in
/path/to/venv/lib/python3.6/site-packages (from pytest->pytest-nice==0.1.0)
Requirement already satisfied: setuptools in
/path/to/venv/lib/python3.6/site-packages (from pytest->pytest-nice==0.1.0)
Building wheels for collected packages: pytest-nice
Running setup.py bdist_wheel for pytest-nice ... done
 ...
Successfully built pytest-nice
Installing collected packages: pytest-nice
Successfully installed pytest-nice-0.1.0
```

Now that it's installed, let's run the tests:

```
$ pytest -v
====================== test session starts ======================
plugins: nice-0.1.0
collected 7 items

tests/test_nice.py::test_pass_fail PASSED
tests/test_nice.py::test_with_nice PASSED
tests/test_nice.py::test_with_nice_verbose PASSED
tests/test_nice.py::test_not_nice_verbose PASSED
tests/test_nice.py::test_header PASSED
tests/test_nice.py::test_header_not_nice PASSED
tests/test_nice.py::test_help_message PASSED

=================== 7 passed in 0.34 seconds ====================
```

Yay! All the tests pass. We can uninstall it just like any other Python package or pytest plugin:

```
$ pip uninstall pytest-nice
Uninstalling pytest-nice-0.1.0:
/path/to/venv/lib/python3.6/site-packages/pytest-nice.egg-link
 ...
Proceed (y/n)? y
Successfully uninstalled pytest-nice-0.1.0
```

A great way to learn more about plugin testing is to look at the tests contained in other pytest plugins available through PyPI.

Creating a Distribution

Believe it or not, we are almost done with our plugin. From the command line, we can use this setup.py file to create a distribution:

```
$ cd /path/to/code/ch5/pytest-nice
$ python setup.py sdist
running sdist
running egg_info
creating pytest_nice.egg-info
...
running check
creating pytest-nice-0.1.0
...
creating dist
Creating tar archive
...
$ ls dist
pytest-nice-0.1.0.tar.gz
```

(Note that sdist stands for “source distribution.”)

Within pytest-nice, a dist directory contains a new file called pytest-nice-0.1.0.tar.gz. This file can now be used anywhere to install our plugin, even in place:

```
$ pip install dist/pytest-nice-0.1.0.tar.gz
Processing ./dist/pytest-nice-0.1.0.tar.gz
...
Installing collected packages: pytest-nice
Successfully installed pytest-nice-0.1.0
```

However, you can put your .tar.gz files anywhere you’ll be able to get at them to use and share.

Distributing Plugins Through a Shared Directory

pip already supports installing packages from shared directories, so all we have to do to distribute our plugin through a shared directory is pick a location we can remember and put the .tar.gz files for our plugins there. Let’s say we put pytest-nice-0.1.0.tar.gz into a directory called myplugins.

To install pytest-nice from myplugins:

```
$ pip install --no-index --find-links myplugins pytest-nice
```

The --no-index tells pip to not go out to PyPI to look for what you want to install. The --find-links myplugins tells PyPI to look in myplugins for packages to install. And of course, pytest-nice is what we want to install.

If you've done some bug fixes and there are newer versions in myplugins, you can upgrade by adding --upgrade:

```
$ pip install --upgrade --no-index --find-links myplugins pytest-nice
```

This is just like any other use of pip, but with the --no-index --find-links myplugins added.

Distributing Plugins Through PyPI

If you want to share your plugin with the world, there are a few more steps we need to do. Actually, there are quite a few more steps. However, because this book isn't focused on contributing to open source, I recommend checking out the thorough instruction found in the Python Packaging User Guide.[16]

When you are contributing a pytest plugin, another great place to start is by using the cookiecutter-pytest-plugin[17]:

```
$ pip install cookiecutter
$ cookiecutter https://github.com/pytest-dev/cookiecutter-pytest-plugin
```

This project first asks you some questions about your plugin. Then it creates a good directory for you to explore and fill in with your code. Walking through this is beyond the scope of this book; however, please keep this project in mind. It is supported by core pytest folks, and they will make sure this project stays up to date.

Exercises

In ch4/cache/test_slower.py, there is an autouse fixture called check_duration(). You used it in the Chapter 4 exercises as well. Now, let's make a plugin out of it.

1. Create a directory named pytest-slower that will hold the code for the new plugin, similar to the directory described in *Creating an Installable Plugin*.
2. Fill out all the files of the directory to make pytest-slower an installable plugin.
3. Write some test code for the plugin.
4. Take a look at the Python Package Index[18] and search for "pytest-." Find a pytest plugin that looks interesting to you.
5. Install the plugin you chose and try it out on Tasks tests.

What’s Next

You’ve used conftest.py a lot so far in this book. There are also configuration files that affect how pytest runs, such as pytest.ini. In the next chapter, you’ll run through the different configuration files and learn what you can do there to make your testing life easier.

Footnotes

[11] http://doc.pytest.org/en/latest/_modules/_pytest/hookspec.html

[12] https://docs.pytest.org/en/latest/writing_plugins.html

[13] http://python-packaging.readthedocs.io

[14] https://www.pypa.io

[15] https://docs.pytest.org/en/latest/writing_plugins.html#_pytest.pytester.RunResult

[16] https://packaging.python.org/distributing

[17] https://github.com/pytest-dev/cookiecutter-pytest-plugin

[18] https://pypi.python.org/pypi

Chapter 6
Configuration

So far in this book, I've talked about the various non-test files that affect pytest mostly in passing, with the exception of conftest.py, which I covered quite thoroughly in Chapter 5, *Plugins*. In this chapter, we'll take a look at the configuration files that affect pytest, discuss how pytest changes its behavior based on them, and make some changes to the configuration files of the Tasks project.

Understanding pytest Configuration Files

Before I discuss how you can alter pytest's default behavior, let's run down all of the non-test files in pytest and specifically who should care about them. Everyone should know about these:

- pytest.ini: This is the primary pytest configuration file that allows you to change default behavior. Since there are quite a few configuration changes you can make, a big chunk of this chapter is about the settings you can make in pytest.ini.

- conftest.py: This is a local plugin to allow hook functions and fixtures for the directory where the conftest.py file exists and all subdirectories. conftest.py files are covered Chapter 5, *Plugins*.

- __init__.py: When put into every test subdirectory, this file allows you to have identical test filenames in multiple test directories. We'll look at an example of what can go wrong without __init__.py files in test directories in *Avoiding Filename Collisions*.

If you use tox, you'll be interested in:

- tox.ini: This file is similar to pytest.ini, but for tox. However, you can put your pytest configuration here instead of having both a tox.ini and a pytest.ini file, saving you one configuration file. Tox is covered in Chapter 7, *Using pytest with Other Tools*.

If you want to distribute a Python package (like Tasks), this file will be of interest:

- setup.cfg: This is a file that's also in ini file format and affects the behavior of setup.py. It's possible to add a couple of lines to setup.py to allow you to run python setup.py test and have it run all of your pytest tests. If you are distributing a package, you may already have a setup.cfg file, and you can use that file to store pytest configuration. You'll see how in Appendix 4, *Packaging and Distributing Python Projects*.

Regardless of which file you put your pytest configuration in, the format will mostly be the same.

For pytest.ini:

ch6/format/pytest.ini

```
[pytest]
addopts = -rsxX -l --tb=short --strict
xfail_strict = true
... more options ...
```

For tox.ini:

ch6/format/tox.ini

```
... tox specific stuff ...


[pytest]
```

```
addopts = -rsxX -l --tb=short --strict
xfail_strict = true
... more options ...
```

For setup.cfg:

ch6/format/setup.cfg

```
... packaging specific stuff ...

[tool:pytest]
addopts = -rsxX -l --tb=short --strict
xfail_strict = true
... more options ...
```

The only difference is that the section header for setup.cfg is [tool:pytest] instead of [pytest].

List the Valid ini-file Options with pytest –help

You can get a list of all the valid settings for pytest.ini from pytest --help:

```
$ pytest --help
...
[pytest] ini-options in the first pytest.ini|tox.ini|setup.cfg file found:

markers (linelist) markers for test functions
norecursedirs (args) directory patterns to avoid for recursion
testpaths (args) directories to search for tests when no files or
directories are given in the command line.
usefixtures (args) list of default fixtures to be used with this project
python_files (args) glob-style file patterns for Python test module discovery
python_classes (args) prefixes or glob names for Python test class discovery
python_functions (args) prefixes or glob names for Python test function and
method discovery
xfail_strict (bool) default for the strict parameter of xfail markers
when not given explicitly (default: False)
doctest_optionflags (args) option flags for doctests
addopts (args) extra command line options
minversion (string) minimally required pytest version
...
```

You'll look at all of these settings in this chapter, except doctest_optionflags, which is covered in Chapter 7, *Using pytest with Other Tools*.

Plugins Can Add ini-file Options

The previous settings list is not a constant. It is possible for plugins (and conftest.py files) to add ini file options. The added options will be added to the pytest --help output as well.

Now, let’s explore some of the configuration changes we can make with the builtin ini file settings available from core pytest.

Changing the Default Command-Line Options

You've used a lot of command-line options for pytest so far, like -v/--verbose for verbose output and -l/--showlocals to see local variables with the stack trace for failed tests. You may find yourself always using some of those options—or preferring to use them—for a project. If you set addopts in pytest.ini to the options you want, you don't have to type them in anymore. Here's a set I like:

```
[pytest]
addopts = -rsxX -l --tb=short --strict
```

The -rsxX tells pytest to report the reasons for all tests that skipped, xfailed, or xpassed. The -l tells pytest to report the local variables for every failure with the stacktrace. The --tb=short removes a lot of the stack trace. It leaves the file and line number, though. The --strict option disallows markers to be used if they aren't registered in a config file. You'll see how to do that in the next section.

Registering Markers to Avoid Marker Typos

Custom markers, as discussed in *Marking Test Functions*, are great for allowing you to mark a subset of tests to run with a specific marker. However, it's too easy to misspell a marker and end up having some tests marked with @pytest.mark.smoke and some marked with @pytest.mark.somke. By default, this isn't an error. pytest just thinks you created two markers. This can be fixed, however, by registering markers in pytest.ini, like this:

```
[pytest]
markers =
smoke: Run the smoke test functions for tasks project
get: Run the test functions that test tasks.get()
```

With these markers registered, you can now also see them with pytest --markers with their descriptions:

```
$ cd /path/to/code/ch6/b/tasks_proj/tests
$ pytest --markers
@pytest.mark.smoke: Run the smoke test test functions

@pytest.mark.get: Run the test functions that test tasks.get()

@pytest.mark.skip(reason=None): skip the ...

...
```

If markers aren't registered, they won't show up in the --markers list. With them registered, they show up in the list, and if you use --strict, any misspelled or unregistered markers show up as an error. The only difference between ch6/a/tasks_proj and ch6/b/tasks_proj is the contents of the pytest.ini file. It's empty in ch6/a. Let's try running the tests without registering any markers:

```
$ cd /path/to/code/ch6/a/tasks_proj/tests
$ pytest --strict --tb=line
==================== test session starts ====================
collected 45 items / 2 errors

============================ ERRORS ============================
_____________ ERROR collecting func/test_add.py _____________
func/test_add.py:20: in <module>
@pytest.mark.smoke
...
E AttributeError: 'smoke' not a registered marker
_________ ERROR collecting func/test_api_exceptions.py _________
```

```
func/test_api_exceptions.py:30: in <module>
@pytest.mark.smoke
...
E AttributeError: 'smoke' not a registered marker
!!!!!!!!!!! Interrupted: 2 errors during collection !!!!!!!!!!!
=================== 2 error in 0.24 seconds ====================
```

If you use markers in pytest.ini to register your markers, you may as well add --strict to your addopts while you're at it. You'll thank me later. Let's go ahead and add a pytest.ini file to the tasks project:

ch6/b/tasks_proj/tests/pytest.ini

```
[pytest]
addopts = -rsxX -l --tb=short --strict
markers =
    smoke: Run the smoke test test functions
    get: Run the test functions that test tasks.get()
```

This has a combination of flags I prefer over the defaults: -rsxX to report which tests skipped, xfailed, or xpassed, --tb=short for a shorter traceback for failures, and --strict to only allow declared markers. And then a list of markers to allow for the project.

This should allow us to run tests, including the smoke tests:

```
$ cd /path/to/code/ch6/b/tasks_proj/tests
$ pytest --strict -m smoke
==================== test session starts =====================
collected 57 items

func/test_add.py .
func/test_api_exceptions.py ..

==================== 54 tests deselected =====================
========== 3 passed, 54 deselected in 0.06 seconds ===========
```

Requiring a Minimum pytest Version

The minversion setting enables you to specify a minimum pytest version you expect for your tests. For instance, I like to use approx() when testing floating point numbers for “close enough” equality in tests. But this feature didn’t get introduced into pytest until version 3.0. To avoid confusion, I add the following to projects that use approx():

```
[pytest]
minversion = 3.0
```

This way, if someone tries to run the tests using an older version of pytest, an error message appears.

Stopping pytest from Looking in the Wrong Places

Did you know that one of the definitions of “recurse” is to swear at your code twice? Well, no. But, it does mean to traverse subdirectories. In the case of pytest, test discovery traverses many directories recursively. But there are some directories you just know you don’t want pytest looking in.

The default setting for norecurse is ’.* build dist CVS _darcs {arch} and *.egg. Having ’.*’ is a good reason to name your virtual environment ’.venv’, because all directories starting with a dot will not be traversed. However, I have a habit of naming it venv, so I could add that to norecursedirs.

In the case of the Tasks project, you could list src in there also, because having pytest look for test files there would just be a waste of time.

```
[pytest]
norecursedirs = .* venv src *.egg dist build
```

When overriding a setting that already has a useful value, like this setting, it’s a good idea to know what the defaults are and put the ones back you care about, as I did in the previous code with *.egg dist build.

The norecursedirs is kind of a corollary to testpaths, so let’s look at that next.

Specifying Test Directory Locations

Whereas norecursedirs tells pytest where not to look, testpaths tells pytest where to look. testspaths is a list of directories relative to the root directory to look in for tests. It's only used if a directory, file, or nodeid is not given as an argument.

Suppose for the Tasks project we put pytest.ini in the tasks_proj directory instead of under tests:

```
tasks_proj/
├── pytest.ini
├── src
│   └── tasks
│       ├── api.py
│       └── ...
└── tests
    ├── conftest.py
    ├── func
    │   ├── __init__.py
    │   ├── test_add.py
    │   ├── ...
    └── unit
        ├── __init__.py
        ├── test_task.py
        └── ...
```

It could then make sense to put tests in testpaths:

```
[pytest]
testpaths = tests
```

Now, as long as you start pytest from the tasks_proj directory, pytest will only look in tasks_proj/tests. My problem with this is that I often bounce around a test directory during test development and debugging, so I can easily test a subdirectory or file without typing out the whole path. Therefore, for me, this setting doesn't help much with interactive testing.

However, it's great for tests launched from a continuous integration server or from tox. In those cases, you know that the root directory is going to be fixed, and you can list directories relative to that fixed root. These are also the cases where you really want to squeeze your test times, so shaving a bit off of test discovery is awesome.

At first glance, it might seem silly to use both testpaths and norecursedirs at the same time. However, as you've seen, testspaths doesn't help much with interactive testing from different parts of the file system. In those cases, norecursedirs can help. Also, if you have directories with tests that don't contain tests, you could use norecursedirs to avoid those. But really, what would be the point of putting extra directories in tests that don't have tests?

Changing Test Discovery Rules

pytest finds tests to run based on certain test discovery rules. The standard test discovery rules are:

- Start at one or more directory. You can specify filenames or directory names on the command line. If you don't specify anything, the current directory is used.
- Look in the directory and all subdirectories recursively for test modules.
- A test module is a file with a name that looks like test_*.py or *_test.py.
- Look in test modules for functions that start with test_.
- Look for classes that start with Test. Look for methods in those classes that start with test_ but don't have an __init__ method.

These are the standard discovery rules; however, you can change them.

python_classes

The usual test discovery rule for pytest and classes is to consider a class a potential test class if it starts with Test*. The class also can't have an __init__() function. But what if we want to name our test classes <something>Test or <something>Suite? That's where python_classes comes in:

```
[pytest]
python_classes = *Test Test* *Suite
```

This enables us to name classes like this:

```
class DeleteSuite():
def test_delete_1():
...

def test_delete_2():
...


....
```

python_files

Like pytest_classes, python_files modifies the default test discovery rule, which is to look for files that start with test_* or end in *_test.

Let's say you have a custom test framework in which you named all of your test files check_<something>.py. Seems reasonable. Instead of renaming all of your files, just add a line to

pytest.ini like this:

```
[pytest]
python_files = test_* *_test check_*
```

Easy peasy. Now you can migrate your naming convention gradually if you want to, or just leave it as check_*.

python_functions

python_functions acts like the previous two settings, but for test function and method names. The default is test_*. To add check_*—you guessed it—do this:

```
[pytest]
python_functions = test_* check_*
```

Now the pytest naming conventions don’t seem that restrictive, do they? If you don’t like the default naming convention, just change it. However, I encourage you to have a better reason. Migrating hundreds of test files is definitely a good reason.

Disallowing XPASS

Setting xfail_strict = true causes tests marked with @pytest.mark.xfail that don’t fail to be reported as an error. I think this should always be set. For more information on the xfail marker, go to *Marking Tests as Expecting to Fail*.

Avoiding Filename Collisions

The utility of having __init__.py files in every test subdirectory of a project confused me for a long time. However, the difference between having these and not having these is simple. If you have __init__.py files in all of your test subdirectories, you can have the same test filename show up in multiple directories. If you don't, you can't. That's it. That's the effect on you.

Here's an example. Directory a and b both have the file, test_foo.py. It doesn't matter what these files have in them, but for this example, they look like this:

ch6/dups/a/test_foo.py

```
def test_a():
pass
```

ch6/dups/b/test_foo.py

```
def test_b():
pass
```

With a directory structure like this:

```
dups
├── a
│   └── test_foo.py
└── b
    └── test_foo.py
```

These files don't even have the same content, but it's still mucked up. Running them individually will be fine, but running pytest from the dups directory won't work:

```
$ cd /path/to/code/ch6/dups
$ pytest a
================== test session starts ==================
collected 1 items

a/test_foo.py .

=============== 1 passed in 0.01 seconds ================
$ pytest b
================== test session starts ==================
collected 1 items

b/test_foo.py .

=============== 1 passed in 0.01 seconds ================
```

```
$ pytest
================== test session starts ==================
collected 1 items / 1 errors

======================== ERRORS ========================
_____________ ERROR collecting b/test_foo.py _____________
import file mismatch:
imported module 'test_foo' has this __file__ attribute:
/path/to/code/ch6/dups/a/test_foo.py
which is not the same as the test file we want to collect:
/path/to/code/ch6/dups/b/test_foo.py
HINT: remove __pycache__ / .pyc files and/or use a unique basename
for your test file modules
!!!!!!!! Interrupted: 1 errors during collection !!!!!!!!
================ 1 error in 0.15 seconds ================
```

That error message doesn't really make it clear what went wrong.

To fix this test, just add empty __init__.py files in the subdirectories. Here, the example directory dups_fixed is the same as dups, but with __init__.py files added:

```
dups_fixed/
├── a
│   ├── __init__.py
│   └── test_foo.py
└── b
    ├── __init__.py
    └── test_foo.py
```

Now, let's try this again from the top level in dups_fixed:

```
$ cd /path/to/code/ch6/dups_fixed
$ pytest
================== test session starts ==================
collected 2 items

a/test_foo.py .
b/test_foo.py .

================ 2 passed in 0.01 seconds ================
```

There, all better. You might say to yourself that you'll never have duplicate filenames, so it doesn't matter. That's fine. But projects grow and test directories grow, and do you really want to wait until it happens to you before you fix it? I say just put those files in there as a habit and

don’t worry about it again.

Exercises

In Chapter 5, *Plugins*, you created a plugin called pytest-nice that included a --nice command-line option. Let's extend that to include a pytest.ini option called nice.

1. Add the following line to the pytest_addoption hook function in pytest_nice.py:

   parser.addini('nice', type='bool', help='Turn failures into opportunities.')

2. The places in the plugin that use getoption() will have to also call getini('nice'). Make those changes.

3. Manually test this by adding nice to a pytest.ini file.

4. Don't forget the plugin tests. Add a test to verify that the setting 'nice' from pytest.ini works correctly.

5. Add the tests to the plugin tests directory. You'll need to look up some extra pytester functionality.[19]

What’s Next

While pytest is extremely powerful on its own—especially so with plugins—it also integrates well with other software development and software testing tools. In the next chapter, you’ll look at using pytest in conjunction with other powerful testing tools.

Footnotes

[19]

https://docs.pytest.org/en/latest/_modules/_pytest/pytester.html#Testdir

Chapter 7
Using pytest with Other Tools

You don’t usually use pytest on its own, but rather in a testing environment with other tools. This chapter looks at other tools that are often used in combination with pytest for effective and efficient testing. While this is by no means an exhaustive list, the tools discussed here give you a taste of the power of mixing pytest with other tools.

pdb: Debugging Test Failures

The pdb module is the Python debugger in the standard library. You use --pdb to have pytest start a debugging session at the point of failure. Let's look at pdb in action in the context of the Tasks project.

In *Parametrizing Fixtures*, we left the Tasks project with a few failures:

```
$ cd /path/to/code/ch3/c/tasks_proj
$ pytest --tb=no -q
.....................................FF.FFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF.FFF.....
42 failed, 54 passed in 4.74 seconds
```

Before we look at how pdb can help us debug this test, let's take a look at the pytest options available to help speed up debugging test failures, which we first looked at in *Using Options*:

- --tb=[auto/long/short/line/native/no]: Controls the traceback style.
- -v / --verbose: Displays all the test names, passing or failing.
- -l / --showlocals: Displays local variables alongside the stacktrace.
- -lf / --last-failed: Runs just the tests that failed last.
- -x / --exitfirst: Stops the tests session with the first failure.
- --pdb: Starts an interactive debugging session at the point of failure.

Installing MongoDB

As mentioned in Chapter 3, *pytest Fixtures*, running the MongoDB tests requires installing MongoDB and pymongo. I've been testing with the Community Server edition found at https://www.mongodb.com/download-center. pymongo is installed with pip: pip install pymongo. However, this is the last example in the book that uses MongoDB. To try out the debugger without using MongoDB, you could run the pytest commands from code/ch2/, as this directory also contains a few failing tests.

We just ran the tests from code/ch3/c to see that some of them were failing. We didn't see the tracebacks or the test names because --tb=no turns off tracebacks, and we didn't have --verbose turned on. Let's re-run the failures (at most three of them) with verbose:

```
$ pytest --tb=no --verbose --lf --maxfail=3
==================== test session starts ====================
run-last-failure: rerun last 42 failures
collected 96 items

tests/func/test_add.py::test_add_returns_valid_id[mongo] FAILED
```

```
tests/func/test_add.py::test_added_task_has_id_set[mongo] FAILED
tests/func/test_add_variety.py::test_add_1[mongo] FAILED

!!!!!!!!!!!! Interrupted: stopping after 3 failures !!!!!!!!!!!!
====================== 54 tests deselected ======================
=========== 3 failed, 54 deselected in 3.14 seconds ============
```

Now we know which tests are failing. Let's look at just one of them by using -x, including the traceback by not using --tb=no, and showing the local variables with -l:

```
$ pytest -v --lf -l -x
====================== test session starts ======================
run-last-failure: rerun last 42 failures
collected 96 items

tests/func/test_add.py::test_add_returns_valid_id[mongo] FAILED

============================ FAILURES ============================
_________________ test_add_returns_valid_id[mongo] _________________

tasks_db = None

def test_add_returns_valid_id(tasks_db):
"""tasks.add(<valid task>) should return an integer."""
 # GIVEN an initialized tasks db
 # WHEN a new task is added
 # THEN returned task_id is of type int
new_task = Task('do something')
task_id = tasks.add(new_task)
> assert isinstance(task_id, int)
E AssertionError: assert False
E + where False = isinstance(ObjectId('59783baf8204177f24cb1b68'), int)

new_task = Task(summary='do something', owner=None, done=False, id=None)
task_id = ObjectId('59783baf8204177f24cb1b68')
tasks_db = None

tests/func/test_add.py:16: AssertionError
!!!!!!!!!!!! Interrupted: stopping after 1 failures !!!!!!!!!!!!
====================== 54 tests deselected ======================
=========== 1 failed, 54 deselected in 2.47 seconds ============
```

Quite often this is enough to understand the test failure. In this particular case, it's pretty clear

that task_id is not an integer—it's an instance of ObjectId. ObjectId is a type used by MongoDB for object identifiers within the database. My intention with the tasksdb_pymongo.py layer was to hide particular details of the MongoDB implementation from the rest of the system. Clearly, in this case, it didn't work.

However, we want to see how to use pdb with pytest, so let's pretend that it wasn't obvious why this test failed. We can have pytest start a debugging session and start us right at the point of failure with --pdb:

```
$ pytest -v --lf -x --pdb
==================== test session starts ====================
run-last-failure: rerun last 42 failures
collected 96 items

tests/func/test_add.py::test_add_returns_valid_id[mongo] FAILED
>>>>>>>>>>>>>>>>>>>>>>>>>> traceback >>>>>>>>>>>>>>>>>>>>>>>>>>>

tasks_db = None

def test_add_returns_valid_id(tasks_db):
"""tasks.add(<valid task>) should return an integer."""
 # GIVEN an initialized tasks db
 # WHEN a new task is added
 # THEN returned task_id is of type int
new_task = Task('do something')
task_id = tasks.add(new_task)
> assert isinstance(task_id, int)
E AssertionError: assert False
E + where False = isinstance(ObjectId('59783bf48204177f2a786893'), int)

tests/func/test_add.py:16: AssertionError
>>>>>>>>>>>>>>>>>>>>>>>>> entering PDB >>>>>>>>>>>>>>>>>>>>>>>>>
> /path/to/code/ch3/c/tasks_proj/tests/func/test_add.py(16)
> test_add_returns_valid_id()
-> assert isinstance(task_id, int)
(Pdb)
```

Now that we are at the (Pdb) prompt, we have access to all of the interactive debugging features of pdb. When looking at failures, I regularly use these commands:

- p/print expr: Prints the value of exp.
- pp expr: Pretty prints the value of expr.
- l/list: Lists the point of failure and five lines of code above and below.

- l/list begin,end: Lists specific line numbers.
- a/args: Prints the arguments of the current function with their values. (This is helpful when in a test helper function.)
- u/up: Moves up one level in the stack trace.
- d/down: Moves down one level in the stack trace.
- q/quit: Quits the debugging session.

Other navigation commands like step and next aren't that useful since we are sitting right at an assert statement. You can also just type variable names and get the values.

You can use p/print expr similar to the -l/--showlocals option to see values within the function:

```
(Pdb) p new_task
Task(summary='do something', owner=None, done=False, id=None)
(Pdb) p task_id
ObjectId('59783bf48204177f2a786893')
(Pdb)
```

Now you can quit the debugger and continue on with testing.

```
(Pdb) q


!!!!!!!!!!!! Interrupted: stopping after 1 failures !!!!!!!!!!!!
====================== 54 tests deselected ======================
========== 1 failed, 54 deselected in 123.40 seconds ===========
```

If we hadn't used -x, pytest would have opened pdb again at the next failed test. More information about using the pdb module is available in the Python documentation.[20]

Coverage.py: Determining How Much Code Is Tested

Code coverage is a measurement of what percentage of the code under test is being tested by a test suite. When you run the tests for the Tasks project, some of the Tasks functionality is executed with every test, but not all of it. Code coverage tools are great for telling you which parts of the system are being completely missed by tests.

Coverage.py is the preferred Python coverage tool that measures code coverage. You'll use it to check the Tasks project code under test with pytest.

Before you use coverage.py, you need to install it. I'm also going to have you install a plugin called pytest-cov that will allow you to call coverage.py from pytest with some extra pytest options. Since coverage is one of the dependencies of pytest-cov, it is sufficient to install pytest-cov, as it will pull in coverage.py:

```
$ pip install pytest-cov
Collecting pytest-cov
Using cached pytest_cov-2.5.1-py2.py3-none-any.whl
Collecting coverage>=3.7.1 (from pytest-cov)
Using cached coverage-4.4.1-cp36-cp36m-macosx_10_10_x86_64.whl
...
Installing collected packages: coverage, pytest-cov
Successfully installed coverage-4.4.1 pytest-cov-2.5.1
```

Let's run the coverage report on version 2 of Tasks. If you still have the first version of the Tasks project installed, uninstall it and install version 2:

```
$ pip uninstall tasks
Uninstalling tasks-0.1.0:
/path/to/venv/bin/tasks
/path/to/venv/lib/python3.6/site-packages/tasks.egg-link
Proceed (y/n)? y
Successfully uninstalled tasks-0.1.0
$ cd /path/to/code/ch7/tasks_proj_v2
$ pip install -e .
Obtaining file:///path/to/code/ch7/tasks_proj_v2
...
Installing collected packages: tasks
Running setup.py develop for tasks
Successfully installed tasks
$ pip list
...
tasks (0.1.1, /path/to/code/ch7/tasks_proj_v2/src)
...
```

Now that the next version of Tasks is installed, we can run our baseline coverage report:

```
$ cd /path/to/code/ch7/tasks_proj_v2
$ pytest --cov=src
==================== test session starts ====================
plugins: mock-1.6.2, cov-2.5.1
collected 62 items

tests/func/test_add.py ...
tests/func/test_add_variety.py .........................
tests/func/test_add_variety2.py ............
tests/func/test_api_exceptions.py .........
tests/func/test_unique_id.py .
tests/unit/test_cli.py .....
tests/unit/test_task.py ....

---------- coverage: platform darwin, python 3.6.2-final-0 -----------
Name Stmts Miss Cover
------------------------------------------------
src/tasks/__init__.py 2 0 100%
src/tasks/api.py 79 22 72%
src/tasks/cli.py 45 14 69%
src/tasks/config.py 18 12 33%
src/tasks/tasksdb_pymongo.py 74 74 0%
src/tasks/tasksdb_tinydb.py 32 4 88%
------------------------------------------------
TOTAL 250 126 50%


================== 62 passed in 0.47 seconds ==================
```

Since the current directory is tasks_proj_v2 and the source code under test is all within src, adding the option --cov=src generates a coverage report for that specific directory under test only.

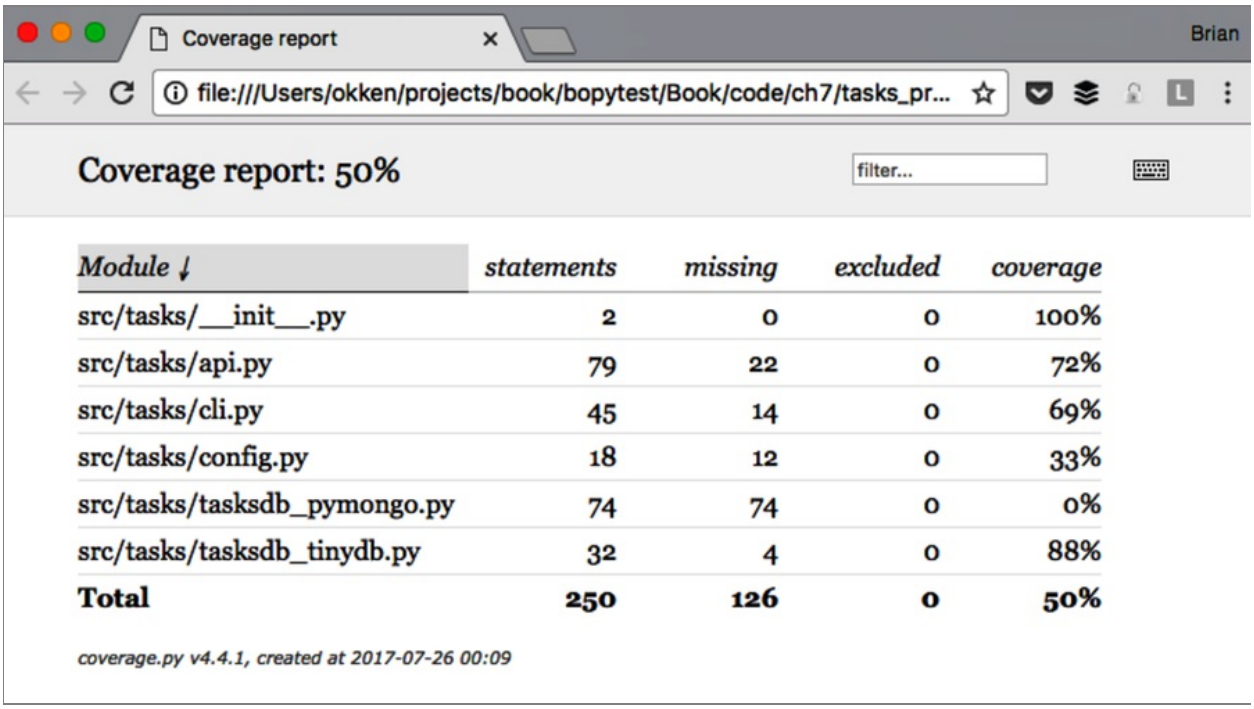
As you can see, some of the files have pretty low, to even 0%, coverage. These are good reminders: tasksdb_pymongo.py is at 0% because we've turned off testing for MongoDB in this version. Some of the others are pretty low. The project will definitely have to put tests in place for all of these areas before it's ready for prime time.

A couple of files I thought would have a higher coverage percentage are api.py and tasksdb_tinydb.py. Let's look at tasksdb_tinydb.py and see what's missing. I find the best way to do that is to use the HTML reports.

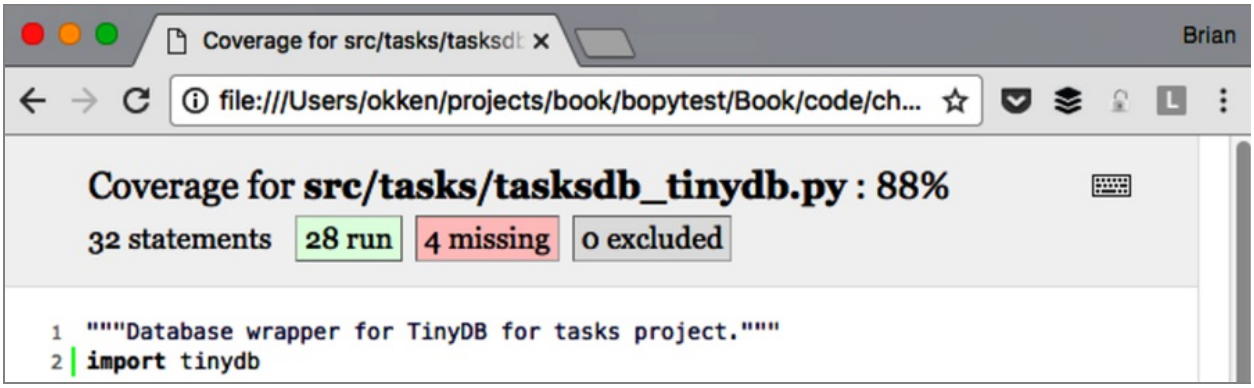
If you run coverage.py again with --cov-report=html, an HTML report is generated:

```
$ pytest --cov=src --cov-report=html
==================== test session starts ====================
plugins: mock-1.6.2, cov-2.5.1
collected 62 items

tests/func/test_add.py ...
tests/func/test_add_variety.py ........................
tests/func/test_add_variety2.py ............
tests/func/test_api_exceptions.py .........
tests/func/test_unique_id.py .
tests/unit/test_cli.py .....
tests/unit/test_task.py ....

---------- coverage: platform darwin, python 3.6.2-final-0 -----------
Coverage HTML written to dir htmlcov


================== 62 passed in 0.45 seconds ==================
```

You can then open htmlcov/index.html in a browser, which shows the output in the following screen:



Clicking on tasksdb_tinydb.py shows a report for the one file. The top of the report shows the percentage of lines covered, plus how many lines were covered and how many are missing, as shown in the following screen:



Scrolling down, you can see the missing lines, as shown in the next screen:

```
31      def list_tasks(self, owner=None):  # type (str) -> list[dict]
32          """Return list of tasks."""
33          if owner is None:
34              return self._db.all()
35          else:
36              return self._db.search(tinydb.Query().owner == owner)
37
38      def count(self):  # type () -> int
39          """Return number of tasks in db."""
40          return len(self._db)
41
42      def update(self, task_id, task):  # type (int, dict) -> ()
43          """Modify task in db with given task_id."""
44          self._db.update(task, eids=[task_id])
45
46      def delete(self, task_id):  # type (int) -> ()
47          """Remove a task from db with given task_id."""
48          self._db.remove(eids=[task_id])
49
50      def delete_all(self):
51          """Remove all tasks from db."""
52          self._db.purge()
53
54      def unique_id(self):  # type () -> int
55          """Return an integer that does not exist in the db."""
56          i = 1
57          while self._db.contains(eids=[i]):
58              i += 1
59          return i
60
```

Even though this screen isn't the complete page for this file, it's enough to tell us that:

1. We're not testing list_tasks() with owner set.
2. We're not testing update() or delete().
3. We may not be testing unique_id() thoroughly enough.

Great. We can put those on our testing to-do list, along with testing the config system.

While code coverage tools are extremely useful, striving for 100% coverage can be dangerous. When you see code that isn't tested, it might mean a test is needed. But it also might mean that there's some functionality of the system that isn't needed and could be removed. Like all software development tools, code coverage analysis does not replace thinking.

Quite a few more options and features of both coverage.py and pytest-cov are available. More

information can be found in the coverage.py[21] and pytest-cov[22] documentation.

mock: Swapping Out Part of the System

The mock package is used to swap out pieces of the system to isolate bits of our code under test from the rest of the system. Mock objects are sometimes called test doubles, spies, fakes, or stubs. Between pytest's own monkeypatch fixture (covered in *Using monkeypatch*) and mock, you should have all the test double functionality you need.

Mocks Are Weird

If this is the first time you've encountered test doubles like mocks, stubs, and spies, it's gonna get real weird real fast. It's fun though, and quite powerful.

The mock package is shipped as part of the Python standard library as unittest.mock as of Python 3.3. In earlier versions, it's available as a separate PyPI-installable package as a rolling backport. What that means is that you can use the PyPI version of mock with Python 2.6 through the latest Python version and get the same functionality as the latest Python mock. However, for use with pytest, a plugin called pytest-mock has some conveniences that make it my preferred interface to the mock system.

For the Tasks project, we'll use mock to help us test the command-line interface. In *Coverage.py: Determining How Much Code Is Tested*, you saw that our cli.py file wasn't being tested at all. We'll start to fix that now. But let's first talk about strategy.

An early decision in the Tasks project was to do most of the functionality testing through api.py. Therefore, it's a reasonable decision that the command-line testing doesn't have to be complete functionality testing. We can have a fair amount of confidence that the system will work through the CLI if we mock the API layer during CLI testing. It's also a convenient decision, allowing us to look at mocks in this section.

The implementation of the Tasks CLI uses the Click third-party command-line interface package. [23] There are many alternatives for implementing a CLI, including Python's builtin argparse module. One of the reasons I chose Click is because it includes a test runner to help us test Click applications. However, the code in cli.py, although hopefully typical of Click applications, is not obvious.

Let's pause and install version 3 of Tasks:

```
$ cd /path/to/code/
$ pip install -e ch7/tasks_proj_v2
...
Successfully installed tasks
```

In the rest of this section, you'll develop some tests for the "list" functionality. Let's see it in action to understand what we're going to test:

```
$ tasks list
 ID owner done summary
 -- ----- ---- -------
$ tasks add 'do something great'
$ tasks add "repeat" -o Brian
$ tasks add "again and again" --owner Okken
$ tasks list
 ID owner done summary
 -- ----- ---- -------
 1 False do something great
 2 Brian False repeat
 3 Okken False again and again
$ tasks list -o Brian
 ID owner done summary
 -- ----- ---- -------
 2 Brian False repeat
$ tasks list --owner Brian
ID owner done summary
-- ----- ---- -------
2 Brian False repeat
```

Looks pretty simple. The tasks list command lists all the tasks with a header. It prints the header even if the list is empty. It prints just the things from one owner if -o or --owner are used. How do we test it? Lots of ways are possible, but we're going to use mocks.

Tests that use mocks are necessarily white-box tests, and we have to look into the code to decide what to mock and where. The main entry point is here:

ch7/tasks_proj_v2/src/tasks/cli.py

```
if __name__ == '__main__':
tasks_cli()
```

That's just a call to tasks_cli():

ch7/tasks_proj_v2/src/tasks/cli.py

```
@click.group(context_settings={'help_option_names': ['-h', '--help']})
@click.version_option(version='0.1.1')
def tasks_cli():
"""Run the tasks application."""
pass
```

Obvious? No. But hold on, it gets better (or worse, depending on your perspective). Here's one of the commands—the list command:

ch7/tasks_proj_v2/src/tasks/cli.py

```
@tasks_cli.command(name="list", help="list tasks")
@click.option('-o', '--owner', default=None,
help='list tasks with this owner')
def list_tasks(owner):
"""
 List tasks in db.

 If owner given, only list tasks with that owner.
 """
formatstr = "{: >4} {: >10} {: >5} {}"
print(formatstr.format('ID', 'owner', 'done', 'summary'))
print(formatstr.format('--', '-----', '----', '-------'))
with _tasks_db():
for t in tasks.list_tasks(owner):
done = 'True' if t.done else 'False'
owner = '' if t.owner is None else t.owner
print(formatstr.format(
t.id, owner, done, t.summary))
```

Once you get used to writing Click code, it's not that bad. I'm not going to explain all of this here, as developing command-line code isn't the focus of the book; however, even though I'm pretty sure I have this code right, there's lots of room for human error. That's why a good set of automated tests to make sure this works correctly is important.

This list_tasks(owner) function depends on a couple of other functions: tasks_db(), which is a context manager, and tasks.list_tasks(owner), which is the API function. We're going to use mock to put fake functions in place for tasks_db() and tasks.list_tasks(). Then we can call the list_tasks method through the command-line interface and make sure it calls the tasks.list_tasks() function correctly and deals with the return value correctly.

To stub tasks_db(), let's look at the real implementation:

ch7/tasks_proj_v2/src/tasks/cli.py

```
@contextmanager
def _tasks_db():
config = tasks.config.get_config()
tasks.start_tasks_db(config.db_path, config.db_type)
yield
tasks.stop_tasks_db()
```

The tasks_db() function is a context manager that retrieves the configuration from tasks.config.get_config(), another external dependency, and uses the configuration to start a connection with the database. The yield releases control to the with block of list_tasks(), and after everything is done, the database connection is stopped.

For the purpose of just testing the CLI behavior up to the point of calling API functions, we

don’t need a connection to an actual database. Therefore, we can replace the context manager with a simple stub:

ch7/tasks_proj_v2/tests/unit/test_cli.py

```
@contextmanager
def stub_tasks_db():
yield
```

Because this is the first time we’ve looked at our test code for test_cli,py, let’s look at this with all of the import statements:

ch7/tasks_proj_v2/tests/unit/test_cli.py

```
from click.testing import CliRunner
from contextlib import contextmanager
import pytest
from tasks.api import Task
import tasks.cli
import tasks.config


@contextmanager
def stub_tasks_db():
yield
```

Those imports are for the tests. The only import needed for the stub is from contextlib import contextmanager.

We’ll use mock to replace the real context manager with our stub. Actually, we’ll use mocker, which is a fixture provided by the pytest-mock plugin. Let’s look at an actual test. Here’s a test that calls tasks list:

ch7/tasks_proj_v2/tests/unit/test_cli.py

```
def test_list_no_args(mocker):
mocker.patch.object(tasks.cli, '_tasks_db', new=stub_tasks_db)
mocker.patch.object(tasks.cli.tasks, 'list_tasks', return_value=[])
runner = CliRunner()
runner.invoke(tasks.cli.tasks_cli, ['list'])
tasks.cli.tasks.list_tasks.assert_called_once_with(None)
```

The mocker fixture is provided by pytest-mock as a convenience interface to unittest.mock. The first line, mocker.patch.object(tasks.cli, ’tasks_db’, new=stub_tasks_db), replaces the tasks_db() context manager with our stub that does nothing.

The second line, mocker.patch.object(tasks.cli.tasks, ’list_tasks’, return_value=[]), replaces any calls to tasks.list_tasks() from within tasks.cli to a default MagicMock object with a return value of an empty list. We can use this object later to see if it was called correctly. The MagicMock class is a flexible subclass of unittest.Mock with reasonable default behavior and the ability to

specify a return value, which is what we are using in this example. The Mock and MagicMock classes (and others) are used to mimic the interface of other code with introspection methods built in to allow you to ask them how they were called.

The third and fourth lines of test_list_no_args() use the Click CliRunner to do the same thing as calling tasks list on the command line.

The final line uses the mock object to make sure the API call was called correctly. The assert_called_once_with() method is part of unittest.mock.Mock objects, which are all listed in the Python documentation.[24]

Let's look at an almost identical test function that checks the output:

ch7/tasks_proj_v2/tests/unit/test_cli.py

```
@pytest.fixture()
def no_db(mocker):
mocker.patch.object(tasks.cli, '_tasks_db', new=stub_tasks_db)


def test_list_print_empty(no_db, mocker):
mocker.patch.object(tasks.cli.tasks, 'list_tasks', return_value=[])
runner = CliRunner()
result = runner.invoke(tasks.cli.tasks_cli, ['list'])
expected_output = ("  ID owner done summary\n"
" -- ----- ---- -------\n")
assert result.output == expected_output
```

This time we put the mock stubbing of tasks_db into a no_db fixture so we can reuse it more easily in future tests. The mocking of tasks.list_tasks() is the same as before. This time, however, we are also checking the output of the command-line action through result.output and asserting equality to expected_output.

This assert could have been put in the first test, test_list_no_args, and we could have eliminated the need for two tests. However, I have less faith in my ability to get CLI code correct than other code, so separating the questions of "Is the API getting called correctly?" and "Is the action printing the right thing?" into two tests seems appropriate.

The rest of the tests for the tasks list functionality don't add any new concepts, but perhaps looking at several of these makes the code easier to understand:

ch7/tasks_proj_v2/tests/unit/test_cli.py

```
def test_list_print_many_items(no_db, mocker):
many_tasks = (
Task('write chapter', 'Brian', True, 1),
Task('edit chapter', 'Katie', False, 2),
Task('modify chapter', 'Brian', False, 3),
Task('finalize chapter', 'Katie', False, 4),
```

```
    )
    mocker.patch.object(tasks.cli.tasks, 'list_tasks',
    return_value=many_tasks)
    runner = CliRunner()
    result = runner.invoke(tasks.cli.tasks_cli, ['list'])
    expected_output = (" ID owner done summary\n"
    " -- ----- ---- -------\n"
    " 1 Brian True write chapter\n"
    " 2 Katie False edit chapter\n"
    " 3 Brian False modify chapter\n"
    " 4 Katie False finalize chapter\n")
    assert result.output == expected_output


    def test_list_dash_o(no_db, mocker):
    mocker.patch.object(tasks.cli.tasks, 'list_tasks')
    runner = CliRunner()
    runner.invoke(tasks.cli.tasks_cli, ['list', '-o', 'brian'])
    tasks.cli.tasks.list_tasks.assert_called_once_with('brian')


    def test_list_dash_dash_owner(no_db, mocker):
    mocker.patch.object(tasks.cli.tasks, 'list_tasks')
    runner = CliRunner()
    runner.invoke(tasks.cli.tasks_cli, ['list', '--owner', 'okken'])
    tasks.cli.tasks.list_tasks.assert_called_once_with('okken')
```

Let’s make sure they all work:

```
    $ cd /path/to/code/ch7/tasks_proj_v2
    $ pytest -v tests/unit/test_cli.py
    ================== test session starts ==================
    plugins: mock-1.6.2, cov-2.5.1
    collected 5 items

    tests/unit/test_cli.py::test_list_no_args PASSED
    tests/unit/test_cli.py::test_list_print_empty PASSED
    tests/unit/test_cli.py::test_list_print_many_items PASSED
    tests/unit/test_cli.py::test_list_dash_o PASSED
    tests/unit/test_cli.py::test_list_dash_dash_owner PASSED

    ================ 5 passed in 0.06 seconds ================
```

Yay! They pass.

This was an extremely fast fly-through of using test doubles and mocks. If you want to use mocks in your testing, I encourage you to read up on unittest.mock in the standard library documentation,[25] and about pytest-mock at http://pypi.python.org.[26]

tox: Testing Multiple Configurations

tox is a command-line tool that allows you to run your complete suite of tests in multiple environments. We're going to use it to test the Tasks project in multiple versions of Python. However, tox is not limited to just Python versions. You can use it to test with different dependency configurations and different configurations for different operating systems.

In gross generalities, here's a mental model for how tox works:

tox uses the setup.py file for the package under test to create an installable source distribution of your package. It looks in tox.ini for a list of environments and then for each environment…

1. tox creates a virtual environment in a .tox directory.
2. tox pip installs some dependencies.
3. tox pip installs your package from the sdist in step 1.
4. tox runs your tests.

After all of the environments are tested, tox reports a summary of how they all did.

This makes a lot more sense when you see it in action, so let's look at how to modify the Tasks project to use tox to test Python 2.7 and 3.6. I chose versions 2.7 and 3.6 because they are both already installed on my system. If you have different versions installed, go ahead and change the envlist line to match whichever version you have or are willing to install.

The first thing we need to do to the Tasks project is add a tox.ini file at the same level as setup.py —the top project directory. I'm also going to move anything that's in pytest.ini into tox.ini.

Here's the abbreviated code layout:

```
tasks_proj_v2/
├── ...
├── setup.py
├── tox.ini
├── src
│   └── tasks
│   ├── __init__.py
│   ├── api.py
│   └── ...
└── tests
    ├── conftest.py
    ├── func
    │   ├── __init__.py
    │   ├── test_add.py
    │   └── ...
    └── unit
    ├── __init__.py
```

```
    ├── test_task.py
    └── ...
```

Now, here's what the tox.ini file looks like:

ch7/tasks_proj_v2/tox.ini

```
# tox.ini , put in same dir as setup.py

[tox]
envlist = py27,py36

[testenv]
deps=pytest
commands=pytest

[pytest]
addopts = -rsxX -l --tb=short --strict
markers =
    smoke: Run the smoke test test functions
    get: Run the test functions that test tasks.get()
```

Under [tox], we have envlist = py27,py36. This is a shorthand to tell tox to run our tests using both python2.7 and python3.6.

Under [testenv], the deps=pytest line tells tox to make sure pytest is installed. If you have multiple test dependencies, you can put them on separate lines. You can also specify which version to use.

The commands=pytest line tells tox to run pytest in each environment.

Under [pytest], we can put whatever we normally would want to put into pytest.ini to configure pytest, as discussed in Chapter 6, *Configuration*. In this case, addopts is used to turn on extra summary information for skips, xfails, and xpasses (-rsxX) and turn on showing local variables in stack traces (-l). It also defaults to shortened stack traces (--tb=short) and makes sure all markers used in tests are declared first (--strict). The markers section is where the markers are declared.

Before running tox, you have to make sure you install it:

```
$ pip install tox
```

This can be done within a virtual environment.

Then to run tox, just run, well, tox:

```
$ cd /path/to/code/ch7/tasks_proj_v2
$ tox
GLOB sdist-make: /path/to/code/ch7/tasks_proj_v2/setup.py
```

```
py27 create: /path/to/code/ch7/tasks_proj_v2/.tox/py27
py27 installdeps: pytest
py27 inst: /path/to/code/ch7/tasks_proj_v2/.tox/dist/tasks-0.1.1.zip
py27 installed: click==6.7,funcsigs==1.0.2,mock==2.0.0,
pbr==3.1.1,py==1.4.34,pytest==3.2.1,
pytest-mock==1.6.2,six==1.10.0,tasks==0.1.1,tinydb==3.4.0
py27 runtests: PYTHONHASHSEED='1311894089'
py27 runtests: commands[0] | pytest
================= test session starts ==================
plugins: mock-1.6.2
collected 62 items

tests/func/test_add.py ...
tests/func/test_add_variety.py .........................
tests/func/test_add_variety2.py ............
tests/func/test_api_exceptions.py .........
tests/func/test_unique_id.py .
tests/unit/test_cli.py .....
tests/unit/test_task.py ....

============== 62 passed in 0.25 seconds ===============
py36 create: /path/to/code/ch7/tasks_proj_v2/.tox/py36
py36 installdeps: pytest
py36 inst: /path/to/code/ch7/tasks_proj_v2/.tox/dist/tasks-0.1.1.zip
py36 installed: click==6.7,py==1.4.34,pytest==3.2.1,
pytest-mock==1.6.2,six==1.10.0,tasks==0.1.1,tinydb==3.4.0
py36 runtests: PYTHONHASHSEED='1311894089'
py36 runtests: commands[0] | pytest
================= test session starts ==================
plugins: mock-1.6.2
collected 62 items

tests/func/test_add.py ...
tests/func/test_add_variety.py .........................
tests/func/test_add_variety2.py ............
tests/func/test_api_exceptions.py .........
tests/func/test_unique_id.py .

tests/unit/test_cli.py .....
tests/unit/test_task.py ....

============== 62 passed in 0.27 seconds ===============
```

```
_____ summary _____
  py27: commands succeeded
  py36: commands succeeded
  congratulations :)
```

At the end, we have a nice summary of all the test environments and their outcomes:

```
_____ summary _____
  py27: commands succeeded
  py36: commands succeeded
  congratulations :)
```

Doesn’t that give you a nice, warm, happy feeling? We got a “congratulations” and a smiley face.

tox is much more powerful than what I’m showing here and deserves your attention if you are using pytest to test packages intended to be run in multiple environments. For more detailed information, check out the tox documentation.[27]

Jenkins CI: Automating Your Automated Tests

Continuous integration (CI) systems such as Jenkins[28] are frequently used to launch test suites after each code commit. pytest includes options to generate junit.xml-formatted files required by Jenkins and other CI systems to display test results.

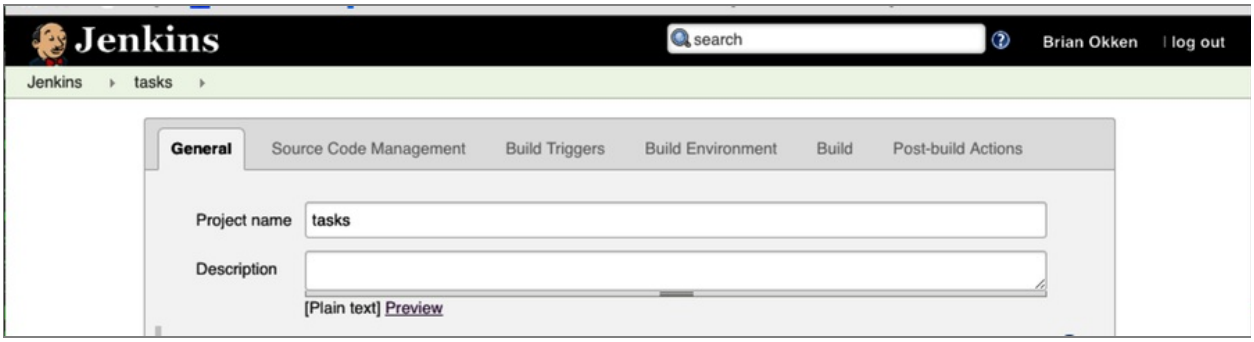
Jenkins is an open source automation server that is frequently used for continuous integration. Even though Python doesn't need to be compiled, it's fairly common practice to use Jenkins or other CI systems to automate the running and reporting of Python projects. In this section, you'll take a look at how the Tasks project might be set up in Jenkins. I'm not going to walk through the Jenkins installation. It's different for every operating system, and instructions are available on the Jenkins website.

When using Jenkins for running pytest suites, there are a few Jenkins plugins that you may find useful. These have been installed for the example:

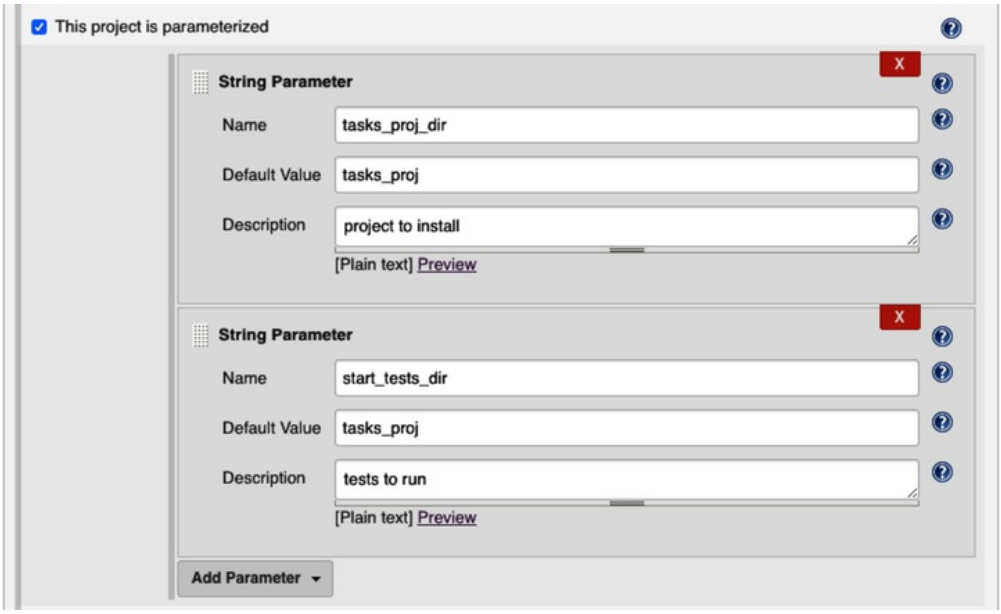
- build-name-setter: This plugin sets the display name of a build to something other than #1, #2, #3, and so on.
- Test Results Analyzer plugin: This plugin shows the history of test execution results in a tabular or graphical format.

You can install plugins by going to the top-level Jenkins page, which is localhost:8080/manage for me as I'm running it locally, then clicking Manage Jenkins -> Manage Plugins -> Available. Search for the plugin you want with the filter box. Check the box for the plugin you want. I usually select "Install without Restart," and then on the Installing Plugins/Upgrades page, I select the box that says, "Restart Jenkins when installation is complete and no jobs are running."

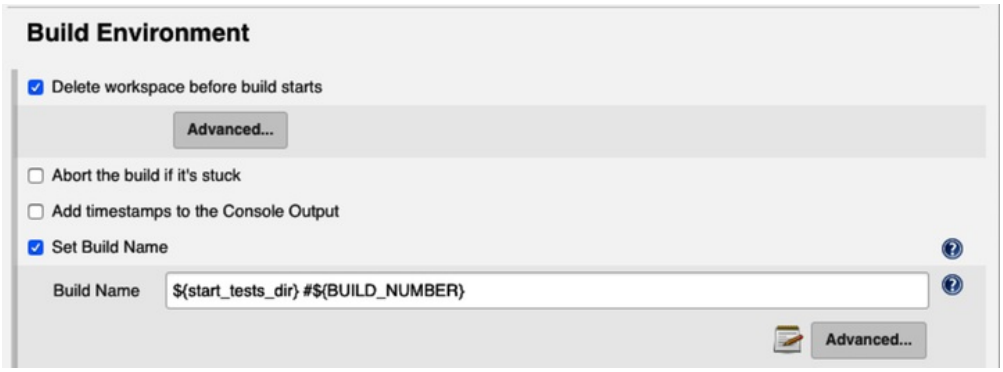
We'll look at a complete configuration in case you'd like to follow along for the Tasks project. The Jenkins project/item is a "Freestyle Project" named "tasks," as shown in the following screen.



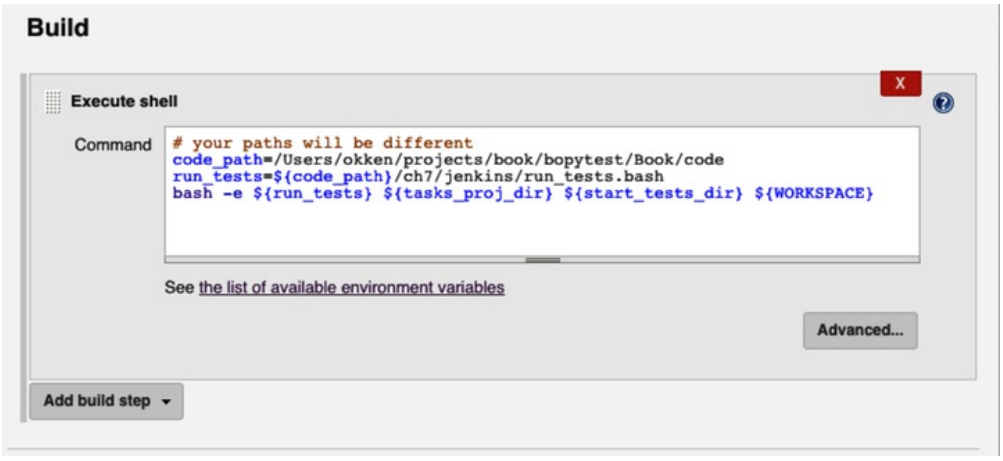
The configuration is a little odd since we're using versions of the Tasks project that look like tasks_proj, tasks_proj_v2, and so on, instead of version control. Therefore, we need to parametrize the project to tell each test session where to install the Tasks project and where to find the tests. We'll use a couple of string parameters, as shown in the next screen, to specify those directories. (Click "This project is parametrized" to get these options available.)



Next, scroll down to Build Environment, and select “Delete workspace before build starts” and Set Build Name. Set the name to ${start_tests_dir} #${BUILD_NUMBER}, as shown in the next screen.



Next are the Build steps. On a Mac or Unix-like systems, select Add build step-> Execute shell. On Windows, select Add build step->Execute Windows batch command. Since I’m on a Mac, I used an Execute shell block to call a script, as shown here:



The content of the text box is:

```
# your paths will be different
code_path=/Users/okken/projects/book/bopytest/Book/code
run_tests=${code_path}/ch7/jenkins/run_tests.bash
bash -e ${run_tests} ${tasks_proj_dir} ${start_tests_dir} ${WORKSPACE}
```

We use a script instead of putting all of this code into the execute block in Jenkins so that any changes can be tracked with revision control. Here's the script:

ch7/jenkins/run_tests.bash

```
#!/bin/bash

# your paths will be different
top_path=/Users/okken/projects/book/bopytest/Book
code_path=${top_path}/code
venv_path=${top_path}/venv
tasks_proj_dir=${code_path}/$1
start_tests_dir=${code_path}/$2
results_dir=$3

# click and Python 3,
# from http://click.pocoo.org/5/python3/
export LC_ALL=en_US.utf-8
export LANG=en_US.utf-8

# virtual environment
source ${venv_path}/bin/activate

# install project
pip install -e ${tasks_proj_dir}

# run tests
cd ${start_tests_dir}
pytest --junit-xml=${results_dir}/results.xml
```

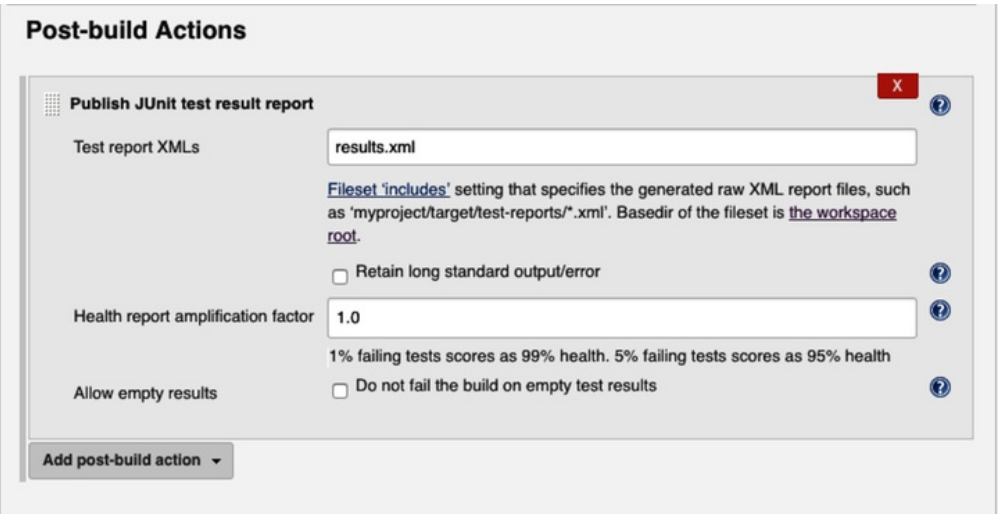
The bottom line has pytest --junit-xml=${results_dir}/results.xml. The --junit-xml flag is the only thing needed to produce the junit.xml format results file Jenkins needs.

There are other options:

```
$ pytest --help | grep junit
--junit-xml=path create junit-xml style report file at given path.
--junit-prefix=str prepend prefix to classnames in junit-xml output
junit_suite_name (string) Test suite name for JUnit report
```

The --junit-prefix can be used as a prefix for every test. This is useful when using tox and you want to separate the different environment results. junit_suite_name is a config file option that you can set in the [pytest] section of pytest.ini or tox.ini. Later we'll see that the results will have from (pytest) in them. To change pytest to something else, use junit_suite_name.

Next, we'll add a post-build action: Add post-build action->Publish Junit test result report. Fill in the Test report XMLs with results.xml, as shown in the next screen.

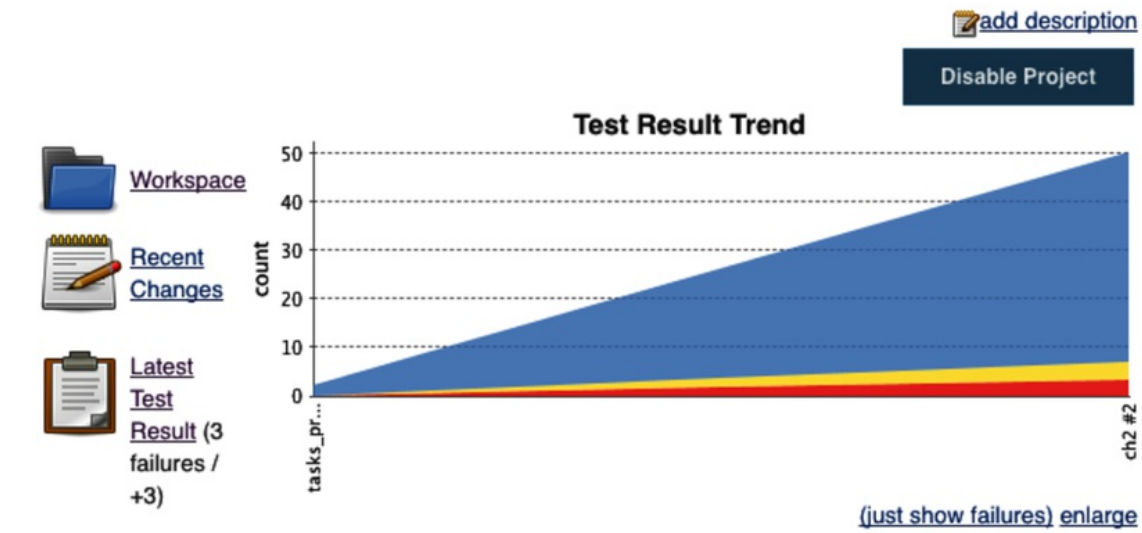


That's it! Now we can run tests through Jenkins. Here are the steps:

1. Click Save.
2. Go to the top project.
3. Click "Build with Parameters."
4. Select your directories and click Build.
5. When it's done, hover over the title next to the ball in Build History and select Console Output from the drop-down menu that appears. (Or click the build name and select Console Output.)
6. Look at the output and try to figure out what went wrong.

You may be able to skip steps 5 and 6, but I never do. I've never set up a Jenkins job and had it work the first time. There are usually directory permission problems or path issues or typos in my script, and so on.

Before we look at the results, let's run one more version to make it interesting. Click "Build with Parameters" again. This time, keep the same project directory, but set ch2 as the start_tests_dir, and click Build. After a refresh of the project top view, you should see the following screen:



Click inside the graph or on the “Latest Test Result” link to see an overview of the test session, with “+” icons to expand for test failures.

Clicking on any of the failing test names shows you the individual test failure information, as shown in the next screen. This is where you see the “(from pytest)” as part of the test name. This is what’s controlled by the junit_suite_name in a config file.

Failed

tasks_proj.tests.func.test_unique_id_1.test_unique_id (from pytest)

Failing for the past 1 build (Since #2)
Took 2 ms.
add description

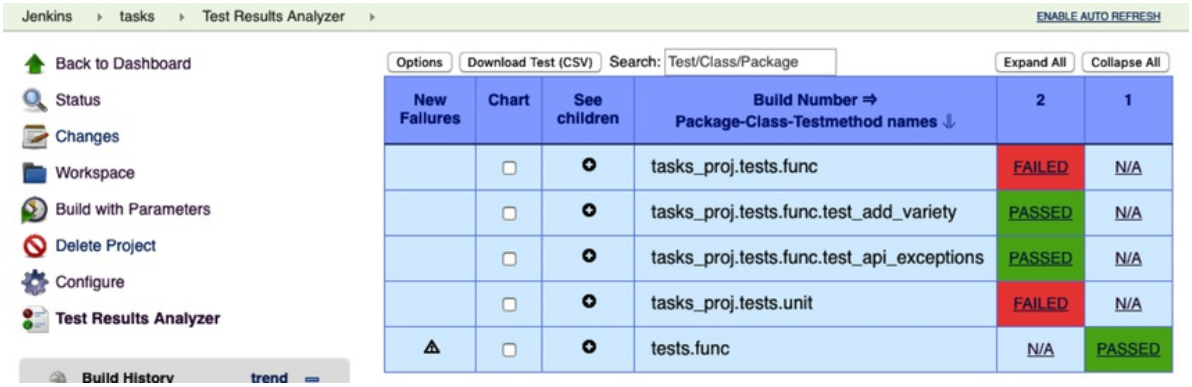
Error Message

```
assert 1 != 1
```

Stacktrace

```
def test_unique_id():
        id_1 = tasks.unique_id()
        id_2 = tasks.unique_id()
>       assert id_1 != id_2
E       assert 1 != 1

tasks_proj/tests/func/test_unique_id_1.py:9: AssertionError
```

Going back to Jenkins > tasks, you can click on Test Results Analyzer to see a view that lists which tests haven't been run for different sessions, along with the pass/fail status (see the following screen):



You've seen how to run pytest suites with virtual environments from Jenkins, but there are quite a few other topics to explore around using pytest and Jenkins together. You can test multiple environments with Jenkins by either setting up separate Jenkins tasks for each environment, or by having Jenkins call tox directly. There's also a nice plugin called Cobertura that is able to display coverage data from coverage.py. Check out the Jenkins documentation[29] for more information.

unittest: Running Legacy Tests with pytest

unittest is the test framework built into the Python standard library. Its purpose is to test Python itself, but it is often used for project testing, too. pytest works as a unittest runner, and can run both pytest and unittest tests in the same session.

Let's pretend that when the Tasks project started, it used unittest instead of pytest for testing. And perhaps there are a lot of tests already written. Fortunately, you can use pytest to run unittest-based tests. This might be a reasonable option if you are migrating your testing effort from unittest to pytest. You can leave all the old tests as unittest, and write new ones in pytest. You can also gradually migrate older tests as you have time, or as changes are needed. There are a couple of issues that might trip you up in the migration, however, and I'll address some of those here. First, let's look at a test written for unittest:

ch7/unittest/test_delete_unittest.py

```
import unittest
import shutil
import tempfile
import tasks
from tasks import Task


def setUpModule():
"""Make temp dir, initialize DB."""
global temp_dir
temp_dir = tempfile.mkdtemp()
tasks.start_tasks_db(str(temp_dir), 'tiny')


def tearDownModule():
"""Clean up DB, remove temp dir."""
tasks.stop_tasks_db()
shutil.rmtree(temp_dir)


class TestNonEmpty(unittest.TestCase):

def setUp(self):
tasks.delete_all() # start empty
# add a few items, saving ids
self.ids = []
self.ids.append(tasks.add(Task('One', 'Brian', True)))
self.ids.append(tasks.add(Task('Two', 'Still Brian', False)))
```

```
    self.ids.append(tasks.add(Task('Three', 'Not Brian', False)))

    def test_delete_decreases_count(self):
    # GIVEN 3 items
    self.assertEqual(tasks.count(), 3)
    # WHEN we delete one
    tasks.delete(self.ids[0])
    # THEN count decreases by 1
    self.assertEqual(tasks.count(), 2)
```

The actual test is at the bottom, test_delete_decreases_count(). The rest of the code is there for setup and teardown. This test runs fine in unittest:

```
    $ cd /path/to/code/ch7/unittest
    $ python -m unittest -v test_delete_unittest.py
    test_delete_decreases_count (test_delete_unittest.TestNonEmpty) ... ok

    ----------------------------------------------------------------------
    Ran 1 test in 0.024s

    OK
```

It also runs fine in pytest:

```
    $ pytest -v test_delete_unittest.py
    ========================== test session starts ==========================
    collected 1 items

    test_delete_unittest.py::TestNonEmpty::test_delete_decreases_count PASSED

    ======================== 1 passed in 0.02 seconds ========================
```

This is great if you just want to use pytest as a test runner for unittest. However, our premise is that the Tasks project is migrating to pytest. Let's say we want to migrate tests one at a time and run both unittest and pytest versions at the same time until we are confident in the pytest versions. Let's look at a rewrite for this test and then try running them both:

ch7/unittest/test_delete_pytest.py

```
    import tasks


    def test_delete_decreases_count(db_with_3_tasks):
    ids = [t.id for t in tasks.list_tasks()]
    # GIVEN 3 items
```

```
    assert tasks.count() == 3
    # WHEN we delete one
    tasks.delete(ids[0])
    # THEN count decreases by 1
    assert tasks.count() == 2
```

The fixtures we've been using for the Tasks project, including db_with_3_tasks introduced in *Using Multiple Fixtures*, help set up the database before the test. It's a much smaller file, even though the test function itself is almost identical.

Both tests pass individually:

```
$ pytest -q test_delete_pytest.py
.
1 passed in 0.01 seconds
$ pytest -q test_delete_unittest.py
.
1 passed in 0.02 seconds
```

You can even run them together if—and only if—you make sure the unittest version runs first:

```
$ pytest -v test_delete_unittest.py test_delete_pytest.py
========================== test session starts ==========================
collected 2 items

test_delete_unittest.py::TestNonEmpty::test_delete_decreases_count PASSED
test_delete_pytest.py::test_delete_decreases_count[tiny] PASSED

======================== 2 passed in 0.07 seconds ========================
```

If you run the pytest version first, something goes haywire:

```
$ pytest -v test_delete_pytest.py test_delete_unittest.py
========================== test session starts ==========================
collected 2 items

test_delete_pytest.py::test_delete_decreases_count[tiny] PASSED
test_delete_unittest.py::TestNonEmpty::test_delete_decreases_count PASSED
test_delete_unittest.py::TestNonEmpty::test_delete_decreases_count ERROR

================================ ERRORS =================================
_____ ERROR at teardown of TestNonEmpty.test_delete_decreases_count _____

tmpdir_factory = <_pytest.tmpdir.TempdirFactory object at 0x1038a3128>
```

```
request = <SubRequest 'tasks_db_session'
for <Function 'test_delete_decreases_count[tiny]'>>

@pytest.fixture(scope='session', params=['tiny'])
def tasks_db_session(tmpdir_factory, request):
temp_dir = tmpdir_factory.mktemp('temp')
tasks.start_tasks_db(str(temp_dir), request.param)
yield # this is where the testing happens
> tasks.stop_tasks_db()

conftest.py:11:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

def stop_tasks_db(): # type: () -> None
global _tasksdb
> _tasksdb.stop_tasks_db()
E AttributeError: 'NoneType' object has no attribute 'stop_tasks_db'

../tasks_proj_v2/src/tasks/api.py:104: AttributeError
==================== 2 passed, 1 error in 0.13 seconds ====================
```

You can see that something goes wrong at the end, after both tests have run and passed.

Let’s use --setup-show to investigate further:

```
$ pytest -q --tb=no --setup-show test_delete_pytest.py test_delete_unittest.py

SETUP S tmpdir_factory
SETUP S tasks_db_session (fixtures used: tmpdir_factory)[tiny]
SETUP F tasks_db (fixtures used: tasks_db_session)
SETUP S tasks_just_a_few
SETUP F db_with_3_tasks (fixtures used: tasks_db, tasks_just_a_few)
test_delete_pytest.py::test_delete_decreases_count[tiny]
(fixtures used: db_with_3_tasks, tasks_db, tasks_db_session,
tasks_just_a_few, tmpdir_factory).
TEARDOWN F db_with_3_tasks
TEARDOWN F tasks_db
test_delete_unittest.py::TestNonEmpty::test_delete_decreases_count.
TEARDOWN S tasks_just_a_few
TEARDOWN S tasks_db_session[tiny]
TEARDOWN S tmpdir_factoryE
2 passed, 1 error in 0.08 seconds
```

The session scope teardown fixtures are run after all the tests, including the unittest tests. This stumped me for a bit until I realized that the tearDownModule() in the unittest module was shutting down the connection to the database. The tasks_db_session() teardown from pytest was then trying to do the same thing afterward.

Fix the problem by using the pytest session scope fixture with the unittest tests. This is possible by adding @pytest.mark.usefixtures() decorators at the class or method level:

ch7/unittest/test_delete_unittest_fix.py

```
import pytest
import unittest
import tasks
from tasks import Task


@pytest.mark.usefixtures('tasks_db_session')
class TestNonEmpty(unittest.TestCase):

def setUp(self):
tasks.delete_all() # start empty
# add a few items, saving ids
self.ids = []
self.ids.append(tasks.add(Task('One', 'Brian', True)))
self.ids.append(tasks.add(Task('Two', 'Still Brian', False)))
self.ids.append(tasks.add(Task('Three', 'Not Brian', False)))
def test_delete_decreases_count(self):
# GIVEN 3 items
self.assertEqual(tasks.count(), 3)
# WHEN we delete one
tasks.delete(self.ids[0])
# THEN count decreases by 1
self.assertEqual(tasks.count(), 2)
```

Now both unittest and pytest can cooperate and not run into each other:

```
$ pytest -v test_delete_pytest.py test_delete_unittest_fix.py
==================== test session starts ====================
plugins: mock-1.6.0, cov-2.5.1
collected 2 items

test_delete_pytest.py::test_delete_decreases_count PASSED
test_delete_unittest_fix.py::TestNonEmpty::test_delete_decreases_count PASSED

================= 2 passed in 0.02 seconds =================
```

Note that this is only necessary for session scope resources shared by unittest and pytest. As discussed earlier in *Marking Test Functions*, you can also use pytest markers on unittest tests, such as @pytest.mark.skip() and @pytest.mark.xfail(), and user markers like @pytest.mark.foo().

Going back to the unittest example, we still used setUp() to save the ids of the tasks. Aside from highlighting that getting a list of ids from tasks is obviously an overlooked API method, it also points to a slight issue with using pytst.mark.usefixtures with unittest: we can't pass data from a fixture to a unittest function directly.

However, you can pass it through the cls object that is part of the request object. In the next example, setUp() code has been moved into a function scope fixture that passes the ids through request.cls.ids:

ch7/unittest/test_delete_unittest_fix2.py

```
import pytest
import unittest
import tasks
from tasks import Task


@pytest.fixture()
def tasks_db_non_empty(tasks_db_session, request):
tasks.delete_all() # start empty
# add a few items, saving ids
ids = []
ids.append(tasks.add(Task('One', 'Brian', True)))
ids.append(tasks.add(Task('Two', 'Still Brian', False)))
ids.append(tasks.add(Task('Three', 'Not Brian', False)))
request.cls.ids = ids


@pytest.mark.usefixtures('tasks_db_non_empty')
class TestNonEmpty(unittest.TestCase):

def test_delete_decreases_count(self):
# GIVEN 3 items
self.assertEqual(tasks.count(), 3)
# WHEN we delete one
tasks.delete(self.ids[0])
# THEN count decreases by 1
self.assertEqual(tasks.count(), 2)
```

The test accesses the ids list through self.ids, just like before.

The ability to use marks has a limitation: you cannot use parametrized fixtures with unittest-based tests. However, when looking at the last example with unittest using pytest fixtures, it's not that far from rewriting it in pytest form. Remove the unittest.TestCase base class and change the self.assertEqual() calls to straight assert calls, and you'd be there.

Another limitation with running unittest with pytest is that unittest subtests will stop at the first failure, while unittest will run each subtest, regardless of failures. When all subtests pass, pytest runs all of them. Because you won't see any false-positive results because of this limitation, I consider this a minor difference.

Exercises

1. The test code in ch2 has a few intentionally failing tests. Use --pdb while running these tests. Try it without the -x option and the debugger will open multiple times, once for each failure.
2. Try fixing the code and rerunning tests with --lf --pdb to just run the failed tests and use the debugger. Trying out debugging tools in a casual environment where you can play around and not be worried about deadlines and fixes is important.
3. We noticed lots of missing tests during our coverage exploration. One topic missing is to test tasks.update(). Write some tests of that in the func directory.
4. Run coverage.py. What other tests are missing? If you covered api.py, do you think it would be fully tested?
5. Add some tests to test_cli.py to check the command-line interface for tasks update using mock.
6. Run your new tests (along with all the old ones) against at least two Python versions with tox.
7. Try using Jenkins to graph all the different tasks_proj versions and test permutations in the chapters.

What's Next

You are definitely ready to go out and try pytest with your own projects. And check out the appendixes that follow. If you've made it this far, I'll assume you no longer need help with pip or virtual environments. However, you may not have looked at Appendix 3, *Plugin Sampler Pack*. If you enjoyed this chapter, it deserves your time to at least skim through it. Then, Appendix 4, *Packaging and Distributing Python Projects* provides a quick look at how to share code through various levels of packaging, and Appendix 5, *xUnit Fixtures* covers an alternative style of pytest fixtures that closer resembles traditional xUnit testing tools.

Also, keep in touch! Check out the book's webpage[30] and use the discussion forum[31] and errata[32] pages to help me keep the book lean, relevant, and easy to follow. This book is intended to be a living document. I want to keep it up to date and relevant for every wave of new pytest users.

Footnotes

[20] https://docs.python.org/3/library/pdb.html

[21] https://coverage.readthedocs.io

[22] https://pytest-cov.readthedocs.io

[23] http://click.pocoo.org

[24] https://docs.python.org/dev/library/unittest.mock.html

[25] https://docs.python.org/dev/library/unittest.mock.html

[26] https://pypi.python.org/pypi/pytest-mock

[27] https://tox.readthedocs.io

[28]

https://jenkins.io

[29]

https://wiki.jenkins-ci.org/display/JENKINS/Cobertura+Plugin

[30]

https://pragprog.com/titles/bopytest

[31]

https://forums.pragprog.com/forums/438

[32]

https://pragprog.com/titles/bopytest/errata

Appendix 1
Virtual Environments

Python virtual environments enable you to set up a Python sandbox with its own set of packages separate from the system site-packages in which to work. There are many reasons to use virtual environments, such as if you have multiple services running with the same Python installation, but with different packages and package version requirements. In addition, you might find it handy to keep the dependent package requirements separate for every Python project you work on. Virtual environments let you do that.

The PyPI version of virtualenv works in most environments. As of Python 3.3, the venv virtual environment module is included as part of the standard library. However, some problems with venv have been reported on Ubuntu. Since virtualenv works with Python 3.6 (and as far back as Python 2.6) and on Ubuntu, we'll use virtualenv in this quick overview.

Here's how to set up a virtual environment in macOS and Linux:

```
$ pip install -U virtualenv
$ virtualenv -p /path/to/a/python.exe /path/to/env_name
$ source /path/to/env_name/bin/activate
(env_name) $
... do your work ...
(env_name) $ deactivate
```

You can also drive the process from Python:

```
$ python3.6 -m pip install -U virtualenv
$ python3.6 -m virtualenv env_name
$ source env_name/bin/activate
(env_name) $
... do your work ...
(env_name) $ deactivate
```

In Windows, there's a change to the activate line:

```
C:/> pip install -U virtualenv
C:/> virtualenv -p /path/to/a/python.exe /path/to/env_name
C:/> /path/to/env_name/Scripts/activate.bat
(env_name) C:/>
... do your work ...
(env_name) C:/> deactivate
```

You can do the same trick of driving everything from the Python executable on Windows as well.

In practice, setting up a virtual environment can be done in fewer steps. For example, I don't often update virtualenv if I know I've updated it not too long ago. I also usually put the virtual environment directory, env_name, directly in my project's top directory.

Therefore, the steps are usually just the following:

```
$ cd /path/to/my_proj
$ virtualenv -p $(which python3.6) my_proj_venv
$ source my_proj_venv/bin/activate
(my_proj_venv) $
... do your work ...
(my_proj_venv) $ deactivate
```

I've also seen two additional installation methods that are interesting and could work for you:

1. Put the virtual environment in the project directory (as was done in the previous code), but name the env directory something consistent, such as venv or .venv. The benefit of this is that you can put venv or .venv in your global .gitignore file. The downside is that the environment name hint in the command prompt just tells you that you are using a virtual environment, but not which one.
2. Put all virtual environments into a common directory, such as ~/venvs/. Now the environment names will be different, letting the command prompt be more useful. You also don't need to worry about .gitignore, since it's not in your project tree. Finally, this directory is one place to look if you forget all of the projects you're working on.

Remember, a virtual environment is a directory with links back to the python.exe file and the pip.exe file of the site-wide Python version it's using. But anything you install is installed in the virtual environment directory, and not in the global site-packages directory. When you're done with a virtual environment, you can just delete the directory and it completely disappears.

I've covered the basics and common use case of virtualenv. However, virtualenv is a flexible tool with many options. Be sure to check out virtualenv --help. It may preemptively answer questions you may have about your specific situation. Also, the Python Packaging Authority docs on virtualenv[33] are worth reading if you still have questions.

Footnotes

[33]

https://virtualenv.pypa.io

Appendix 2
pip

pip is the tool used to install Python packages, and it is installed as part of your Python installation. pip supposedly is a recursive acronym that stands for Pip Installs Python or Pip Installs Packages. (Programmers can be pretty nerdy with their humor.) If you have more than one version of Python installed on your system, each version has its own pip package manager.

By default, when you run pip install something, pip will:

1. Connect to the PyPI repository at https://pypi.python.org/pypi.
2. Look for a package called something.
3. Download the appropriate version of something for your version of Python and your system.
4. Install something into the site-packages directory of your Python installation that was used to call pip.

This is a gross understatement of what pip does—it also does cool stuff like setting up scripts defined by the package, wheel caching, and more.

As mentioned, each installation of Python has its own version of pip tied to it. If you're using virtual environments, pip and python are automatically linked to whichever Python version you specified when creating the virtual environment. If you aren't using virtual environments, and you have multiple Python versions installed, such as python3.5 and python3.6, you will probably want to use python3.5 -m pip or python3.6 -m pip instead of pip directly. It works just the same. (For the examples in this appendix, I assume you are using virtual environments so that pip works just fine as-is.)

To check the version of pip and which version of Python it's tied to, use pip --version:

```
(my_env) $ pip --version
pip 9.0.1 from /path/to/code/my_env/lib/python3.6/site-packages (python 3.6)
```

To list the packages you have currently installed with pip, use pip list. If there's something there you don't want anymore, you can uninstall it with pip uninstall something.

```
(my_env) $ pip list
pip (9.0.1)
setuptools (36.2.7)
wheel (0.29.0)
(my_env) $ pip install pytest
...
Installing collected packages: py, pytest
Successfully installed py-1.4.34 pytest-3.2.1
(my_env) $ pip list
```

```
pip (9.0.1)
py (1.4.34)
pytest (3.2.1)
setuptools (36.2.7)
wheel (0.29.0)
```

As shown in this example, pip installs the package you want and also any dependencies that aren't already installed.

pip is pretty flexible. It can install things from other places, such as GitHub, your own servers, a shared directory, or a local package you're developing yourself, and it always sticks the packages in site-packages unless you're using Python virtual environments.

You can use pip to install packages with version numbers from http://pypi.python.org if it's a release version PyPI knows about:

```
$ pip install pytest==3.2.1
```

You can use pip to install a local package that has a setup.py file in it:

```
$ pip install /path/to/package
```

Use ./package_name if you are in the same directory as the package to install it locally:

```
$ cd /path/just/above/package
$ pip install my_package # pip is looking in PyPI for "my_package"
$ pip install ./my_package # now pip looks locally
```

You can use pip to install packages that have been downloaded as zip files or wheels without unpacking them.

You can also use pip to download a lot of files at once using a requirements.txt file:

```
(my_env) $ cat requirements.txt
pytest==3.2.1
pytest-xdist==1.20.0
(my_env) $ pip install -r requirements.txt
...
Successfully installed apipkg-1.4 execnet-1.4.1 pytest-3.2.1 pytest-xdist-1.20.0
```

You can use pip to download a bunch of various versions into a local cache of packages, and then point pip there instead of PyPI to install them into virtual environments later, even when offline.

The following downloads pytest and all dependencies:

```
(my_env) $ mkdir ~/.pipcache
(my_env) $ pip download -d ~/pipcache pytest
```

```
Collecting pytest
Using cached pytest-3.2.1-py2.py3-none-any.whl
Saved /Users/okken/pipcache/pytest-3.2.1-py2.py3-none-any.whl
Collecting py>=1.4.33 (from pytest)
Using cached py-1.4.34-py2.py3-none-any.whl
Saved /Users/okken/pipcache/py-1.4.34-py2.py3-none-any.whl
Collecting setuptools (from pytest)
Using cached setuptools-36.2.7-py2.py3-none-any.whl
Saved /Users/okken/pipcache/setuptools-36.2.7-py2.py3-none-any.whl
Successfully downloaded pytest py setuptools
```

Later, even if you're offline, you can install from the cache:

```
(my_env) $ pip install --no-index --find-links=~/pipcache pytest
Collecting pytest
Collecting py>=1.4.33 (from pytest)
...
Installing collected packages: py, pytest
Successfully installed py-1.4.34 pytest-3.2.1
```

This is great for situations like running tox or continuous integration test suites without needing to grab packages from PyPI. I also use this method to grab a bunch of packages before taking a trip so that I can code on the plane.

The Python Packaging Authority documentation[34] is a great resource for more information on pip.

Footnotes

[34]

https://pip.pypa.io

Appendix 3
Plugin Sampler Pack

Plugins are the booster rockets that enable you to get even more power out of pytest. So many useful plugins are available, it's difficult to pick just a handful to showcase. You've already seen the pytest-cov plugin in *Coverage.py: Determining How Much Code Is Tested*, and the pytest-mock plugin in *mock: Swapping Out Part of the System*. The following plugins give you just a taste of what else is out there.

All of the plugins featured here are available on PyPI and are installed with pip install <plugin-name>.

Plugins That Change the Normal Test Run Flow

The following plugins in some way change how pytest runs your tests.

pytest-repeat: Run Tests More Than Once

To run tests more than once per session, use the pytest-repeat plugin.[35] This plugin is useful if you have an intermittent failure in a test.

Following is a normal test run of tests that start with test_list from ch7/tasks _proj_v2:

```
$ cd /path/to/code/ch7/tasks_proj_v2
$ pip install .
$ pip install pytest-repeat
$ pytest -v -k test_list
==================== test session starts ====================
plugins: repeat-0.4.1, mock-1.6.2
collected 62 items

tests/func/test_api_exceptions.py::test_list_raises PASSED
tests/unit/test_cli.py::test_list_no_args PASSED
tests/unit/test_cli.py::test_list_print_empty PASSED
tests/unit/test_cli.py::test_list_print_many_items PASSED
tests/unit/test_cli.py::test_list_dash_o PASSED
tests/unit/test_cli.py::test_list_dash_dash_owner PASSED

==================== 56 tests deselected ====================
========== 6 passed, 56 deselected in 0.10 seconds ==========
```

With the pytest-repeat plugin, you can use --count to run everything twice:

```
$ pytest --count=2 -v -k test_list
==================== test session starts ====================
plugins: repeat-0.4.1, mock-1.6.2
collected 124 items

tests/func/test_api_exceptions.py::test_list_raises[1/2] PASSED
tests/func/test_api_exceptions.py::test_list_raises[2/2] PASSED
tests/unit/test_cli.py::test_list_no_args[1/2] PASSED
tests/unit/test_cli.py::test_list_no_args[2/2] PASSED
tests/unit/test_cli.py::test_list_print_empty[1/2] PASSED
tests/unit/test_cli.py::test_list_print_empty[2/2] PASSED
```

```
tests/unit/test_cli.py::test_list_print_many_items[1/2] PASSED
tests/unit/test_cli.py::test_list_print_many_items[2/2] PASSED
tests/unit/test_cli.py::test_list_dash_o[1/2] PASSED
tests/unit/test_cli.py::test_list_dash_o[2/2] PASSED
tests/unit/test_cli.py::test_list_dash_dash_owner[1/2] PASSED
tests/unit/test_cli.py::test_list_dash_dash_owner[2/2] PASSED

====================== 112 tests deselected ======================
========== 12 passed, 112 deselected in 0.16 seconds ===========
```

You can repeat a subset of the tests or just one, and even choose to run it 1,000 times overnight if you want to see if you can catch the failure. You can also set it to stop on the first failure.

pytest-xdist: Run Tests in Parallel

Usually all tests run sequentially. And that's just what you want if your tests hit a resource that can only be accessed by one client at a time. However, if your tests do not need access to a shared resource, you could speed up test sessions by running multiple tests in parallel. The pytest-xdist plugin allows you to do that. You can specify multiple processors and run many tests in parallel. You can even push off tests onto other machines and use more than one computer.

Here's a test that takes at least a second to run, with parametrization such that it runs ten times:

appendices/xdist/test_parallel.py

```
import pytest
import time


@pytest.mark.parametrize('x', list(range(10)))
def test_something(x):
time.sleep(1)
```

Notice that it takes over ten seconds to run normally:

```
$ pip install pytest-xdist
$ cd /path/to/code/appendices/xdist
$ pytest test_parallel.py
====================== test session starts ======================
plugins: xdist-1.20.0, forked-0.2
collected 10 items

test_parallel.py ..........

================== 10 passed in 10.07 seconds ==================
```

With the pytest-xdist plugin, you can use -n numprocesses to run each test in a subprocess, and use -n auto to automatically detect the number of CPUs on the system. Here's the same test run on multiple processors:

```
$ pytest -n auto test_parallel.py
==================== test session starts ====================
plugins: xdist-1.20.0, forked-0.2
gw0 [10] / gw1 [10] / gw2 [10] / gw3 [10]
scheduling tests via LoadScheduling
..........
================== 10 passed in 4.27 seconds ==================
```

It's not a silver bullet to speed up your test times by a factor of the number of processors you have—there is overhead time. However, many testing scenarios enable you to run tests in parallel. And when the tests are long, you may as well let them run in parallel to speed up your test time.

The pytest-xdist plugin does a lot more than we've covered here, including the ability to offload tests to different computers altogether, so be sure to read more about the pytest-xdist plugin[36] on PyPI.

pytest-timeout: Put Time Limits on Your Tests

There are no normal timeout periods for tests in pytest. However, if you're working with resources that may occasionally disappear, such as web services, it's a good idea to put some time restrictions on your tests.

The pytest-timeout plugin[37] does just that. It allows you pass a timeout period on the command line or mark individual tests with timeout periods in seconds. The mark overrides the command-line timeout so that the test can be either longer or shorter than the timeout limit.

Let's run the tests from the previous example (with one-second sleeps) with a half-second timeout:

```
$ cd /path/to/code/appendices/xdist
$ pip install pytest-timeout
$ pytest --timeout=0.5 -x test_parallel.py
==================== test session starts ====================
plugins: xdist-1.20.0, timeout-1.2.0, forked-0.2
timeout: 0.5s method: signal
collected 10 items

test_parallel.py F

============================ FAILURES ============================
_____________________ test_something[0] _____________________
```

```
x = 0

@pytest.mark.parametrize('x', list(range(10)))
def test_something(x):
> time.sleep(1)
E Failed: Timeout >0.5s

test_parallel.py:6: Failed
!!!!!!!!!!!! Interrupted: stopping after 1 failures !!!!!!!!!!!!
=================== 1 failed in 0.68 seconds ===================
```

The -x stops testing after the first failure.

Plugins That Alter or Enhance Output

These plugins don't change how test are run, but they do change the output you see.

pytest-instafail: See Details of Failures and Errors as They Happen

Usually pytest displays the status of each test, and then after all the tests are finished, pytest displays the tracebacks of the failed or errored tests. If your test suite is relatively fast, that might be just fine. But if your test suite takes quite a bit of time, you may want to see the tracebacks as they happen, rather than wait until the end. This is the functionality of the pytest-instafail plugin. [38] When tests are run with the --instafail flag, the failures and errors appear right away.

Here's a test with normal failures at the end:

```
$ cd /path/to/code/appendices/xdist
$ pytest --timeout=0.5 --tb=line --maxfail=2 test_parallel.py
================== test session starts ==================
plugins: xdist-1.20.0, timeout-1.2.0, forked-0.2
timeout: 0.5s method: signal
collected 10 items

test_parallel.py FF

======================== FAILURES ========================
/path/to/code/appendices/xdist/test_parallel.py:6: Failed: Timeout >0.5s
/path/to/code/appendices/xdist/test_parallel.py:6: Failed: Timeout >0.5s
!!!!!!!!! Interrupted: stopping after 2 failures !!!!!!!!!!
================ 2 failed in 1.20 seconds ================
```

Here's the same test with --instafail:

```
$ pytest --instafail --timeout=0.5 --tb=line --maxfail=2 test_parallel.py
================== test session starts ==================
plugins: xdist-1.20.0, timeout-1.2.0, instafail-0.3.0, forked-0.2
timeout: 0.5s method: signal
collected 10 items

test_parallel.py F

/path/to/code/appendices/xdist/test_parallel.py:6: Failed: Timeout >0.5s

test_parallel.py F
```

```
/path/to/code/appendices/xdist/test_parallel.py:6: Failed: Timeout >0.5s

!!!!!!!!! Interrupted: stopping after 2 failures !!!!!!!!!!
================ 2 failed in 1.19 seconds ================
```

The --instafail functionality is especially useful for long-running test suites when someone is monitoring the test output. You can read the test failures, including the stack trace, without stopping the test suite.

pytest-sugar: Instafail + Colors + Progress Bar

The pytest-sugar plugin[39] lets you see status not just as characters, but also in color. It also shows failure and error tracebacks during execution, and has a cool progress bar to the right of the shell.

A test without sugar is shown.

```
(venv) $ cd /Users/okken/code/ch7/tasks_proj_v2/
(venv) $ pytest
================================ test session starts ================================
plugins: xdist-1.20.0, mock-1.6.2, forked-0.2
collected 62 items

tests/func/test_add.py ...
tests/func/test_add_variety.py ............................
tests/func/test_add_variety2.py ............
tests/func/test_api_exceptions.py .........
tests/func/test_unique_id.py .
tests/unit/test_cli.py .....
tests/unit/test_task.py ....

============================= 62 passed in 0.24 seconds =============================
```

And here's the test with sugar:

```
(venv) $ pytest
Test session starts (platform: darwin, Python 3.6.2, pytest 3.2.1, pytest-sugar 0.9.0)
rootdir: /Users/okken/code/ch7/tasks_proj_v2, inifile: tox.ini
plugins: xdist-1.20.0, sugar-0.9.0, mock-1.6.2, forked-0.2
================================ test session starts ================================

 tests/func/test_add.py ✓✓✓                                                   5% ▌
 tests/func/test_add_variety.py ✓✓✓✓✓✓✓✓✓✓✓✓✓✓✓✓✓✓✓✓✓✓✓✓✓✓                 50% ▌▌▌▌▌
 tests/func/test_add_variety2.py ✓✓✓✓✓✓✓✓✓✓✓✓                                69% ▌▌▌▌▌▌▌
 tests/func/test_api_exceptions.py ✓✓✓✓✓✓✓✓✓                                  84% ▌▌▌▌▌▌▌▌
 tests/func/test_unique_id.py ✓                                               85% ▌▌▌▌▌▌▌▌
 tests/unit/test_cli.py ✓✓✓✓✓                                                 94% ▌▌▌▌▌▌▌▌▌
 tests/unit/test_task.py ✓✓✓✓                                                100% ▌▌▌▌▌▌▌▌▌▌

Results (0.50s):
      62 passed
```

The checkmarks (or x's for failures) show up as the tests finish. The progress bars grow in real time, too. It's quite satisfying to watch.

pytest-emoji: Add Some Fun to Your Tests

The pytest-emoji plugin[40] allows you to replace all of the test status characters with emojis. You can also change the emojis if you don't like the ones picked by the plugin author. Although this project is perhaps an example of silliness, it's included in this list because it's a small plugin and is a good example on which to base your own plugins.

To demonstrate the emoji plugin in action, following is sample code that produces pass, fail, skip, xfail, xpass, and error. Here it is with normal output and tracebacks turned off:

```
(venv) $ cd /Users/okken/code/appendices/outcomes/
(venv) $ pytest --tb=no
================================= test session starts ==================================
collected 6 items

test_outcomes.py .FxXsE

==== 1 failed, 1 passed, 1 skipped, 1 xfailed, 1 xpassed, 1 error in 0.07 seconds =====
```

Here it is with verbose, -v:

```
(venv) $ pytest --tb=no -v
================================= test session starts ==================================
collected 6 items

test_outcomes.py::test_pass PASSED
test_outcomes.py::test_fail FAILED
test_outcomes.py::test_xfail xfail
test_outcomes.py::test_xpass XPASS
test_outcomes.py::test_skip SKIPPED
test_outcomes.py::test_error ERROR

==== 1 failed, 1 passed, 1 skipped, 1 xfailed, 1 xpassed, 1 error in 0.06 seconds =====
```

Now, here is the sample code with --emoji:

```
(venv) $ pytest --tb=no --emoji
================================= test session starts ==================================
plugins: emoji-0.1.0
collected 6 items

test_outcomes.py 😀😰😞😯😳😡

==== 1 failed, 1 passed, 1 skipped, 1 xfailed, 1 xpassed, 1 error in 0.07 seconds ====
```

And then with both -v and --emoji:

```
(venv) $ pytest --tb=no -v --emoji
================================= test session starts ==================================
plugins: emoji-0.1.0
collected 6 items

test_outcomes.py::test_pass PASSED 😀
test_outcomes.py::test_fail FAILED 😰
test_outcomes.py::test_xfail xfail 😞
test_outcomes.py::test_xpass XPASS 😯
test_outcomes.py::test_skip SKIPPED 😳
test_outcomes.py::test_error ERROR 😡

==== 1 failed, 1 passed, 1 skipped, 1 xfailed, 1 xpassed, 1 error in 0.06 seconds =====
```

It's a pretty fun plugin, but don't dismiss it as silly out of hand; it allows you to change the emoji using hook functions. It's one of the few pytest plugins that demonstrates how to add hook functions to plugin code.

pytest-html: Generate HTML Reports for Test Sessions

The pytest-html plugin[41] is quite useful in conjunction with continuous integration, or in systems with large, long-running test suites. It creates a webpage to view the test results for a pytest session. The HTML report created includes the ability to filter for type of test result: passed, skipped, failed, errors, expected failures, and unexpected passes. You can also sort by test name, duration, or status. And you can include extra metadata in the report, including screenshots or data sets. If you have reporting needs greater than pass vs. fail, be sure to try out pytest-html.

The pytest-html plugin is really easy to start. Just add --html=report_name.html:

```
$ cd /path/to/code/appendices/outcomes
$ pytest --html=report.html
====================== test session starts ======================
metadata: ...
collected 6 items

test_outcomes.py .FxXsE

generated html file: /path/to/code/appendices/outcomes/report.html
============================ ERRORS ============================
_____________________ ERROR at setup of test_error _____________________

@pytest.fixture()
def flaky_fixture():
> assert 1 == 2
E assert 1 == 2

test_outcomes.py:24: AssertionError
========================== FAILURES ==========================
_____________________________ test_fail _____________________________

def test_fail():
> assert 1 == 2
E assert 1 == 2

test_outcomes.py:8: AssertionError
1 failed, 1 passed, 1 skipped, 1 xfailed, 1 xpassed, 1 error in 0.08 seconds
$ open report.html
```

This produces a report that includes the information about the test session and a results and summary page.

The following screen shows the session environment information and summary:

Report generated on 22-Aug-2017 at 22:38:25 by pytest-html v1.15.2

Environment

JAVA_HOME	/Library/Java/JavaVirtualMachines/jdk1.8.0_77.jdk/Contents/Home
Packages	{'pytest': '3.2.1', 'py': '1.4.34', 'pluggy': '0.4.0'}
Platform	Darwin-15.6.0-x86_64-i386-64bit
Plugins	{'metadata': '1.5.0', 'html': '1.15.2'}
Python	3.6.2

Summary

4 tests ran in 0.08 seconds.

(Un)check the boxes to filter the results.

1 passed, 1 skipped, 1 failed, 1 errors, 1 expected failures, 1 unexpected passes

The next screen shows the summary and results:

The report includes JavaScript that allows you to filter and sort, and you can add extra information to the report, including images. If you need to produce reports for test results, this plugin is worth checking out.

Plugins for Static Analysis

Static analysis tools run checks against your code without running it. The Python community has developed some of these tools. The following plugins allow you to run a static analysis tool against both your code under test and the tests themselves in the same session. Static analysis failures show up as test failures.

pytest-pycodestyle, pytest-pep8: Comply with Python's Style Guide

PEP 8 is a style guide for Python code.[42] It is enforced for standard library code, and is used by many—if not most—Python developers, open source or otherwise. The pycodestyle[43] command-line tool can be used to check Python source code to see if it complies with PEP 8. Use the pytest-pycodestyle plugin[44] to run pycodestyle on code in your project, including test code, with the --pep8 flag. The pycodestyle tool used to be called pep8,[45] and pytest-pep8[46] is available if you want to run the legacy tool.

pytest-flake8: Check for Style Plus Linting

While pep8 checks for style, flake8 is a full linter that also checks for PEP 8 style. The flake8 package[47] is a collection of different style and static analysis tools all rolled into one. It includes lots of options, but has reasonable default behavior. With the pytest-flake8 plugin,[48] you can run all of your source code and test code through flake8 and get a failure if something isn't right. It checks for PEP 8, as well as for logic errors. Use the --flake8 option to run flake8 during a pytest session. You can extend flake8 with plugins that offer even more checks, such as flake8-docstrings,[49] which adds pydocstyle checks for PEP 257, Python's docstring conventions.[50]

Plugins for Web Development

Web-based projects have their own testing hoops to jump through. Even pytest doesn't make testing web applications trivial. However, quite a few pytest plugins help make it easier.

pytest-selenium: Test with a Web Browser

Selenium is a project that is used to automate control of a web browser. The pytest-selenium plugin[51] is the Python binding for it. With it, you can launch a web browser and use it to open URLs, exercise web applications, and fill out forms. You can also programmatically control the browser to test a web site or web application.

pytest-django: Test Django Applications

Django is a popular Python-based web development framework. It comes with testing hooks that allow you to test different parts of a Django application without having to use browser-based testing. By default, the builtin testing support in Django is based on unittest. The pytest-django plugin[52] allows you to use pytest instead of unittest to gain all the benefits of pytest. The plugin also includes helper functions and fixtures to speed up test implementation.

pytest-flask: Test Flask Applications

Flask is another popular framework that is sometimes referred to as a microframework. The pytest-flask plugin[53] provides a handful of fixtures to assist in testing Flask applications.

Footnotes

[35] https://pypi.python.org/pypi/pytest-repeat

[36] https://pypi.python.org/pypi/pytest-xdist

[37] https://pypi.python.org/pypi/pytest-timeout

[38] https://pypi.python.org/pypi/pytest-instafail

[39] https://pypi.python.org/pypi/pytest-sugar

[40]

https://pypi.python.org/pypi/pytest-emoji

[41]

https://pypi.python.org/pypi/pytest-html

[42]

https://www.python.org/dev/peps/pep-0008

[43]

https://pypi.python.org/pypi/pycodestyle

[44]

https://pypi.python.org/pypi/pytest-pycodestyle

[45]

https://pypi.python.org/pypi/pep8

[46]

https://pypi.python.org/pypi/pytest-pep8

[47]

https://pypi.python.org/pypi/flake8

[48]

https://pypi.python.org/pypi/pytest-flake8

[49]

https://pypi.python.org/pypi/flake8-docstrings

[50]

https://www.python.org/dev/peps/pep-0257

[51]

https://pypi.python.org/pypi/pytest-selenium

[52]

https://pypi.python.org/pypi/pytest-django

[53]

https://pypi.python.org/pypi/pytest-flask

Appendix 4
Packaging and Distributing Python Projects

The idea of packaging and distribution seems so serious. Most of Python has a rather informal feeling about it, and now suddenly, we're talking "packaging and distribution." However, sharing code is part of working with Python. Therefore, it's important to learn to share code properly with the builtin Python tools. And while the topic is bigger than what I cover here, it needn't be intimidating. All I'm talking about is how to share code in a way that is more traceable and consistent than emailing zipped directories of modules.

This appendix is intended to give you a comfortable understanding of how to set up a project so that it is installable with pip, how to create a source distribution, and how to create a wheel. This is enough for you to be able to share your code locally with a small team. To share it further through PyPI, I'll refer you to some other resources. Let's see how it's done.

Creating an Installable Module

We'll start by learning how to make a small project installable with pip. For a simple one-module project, the minimal configuration is small. I don't recommend you make it quite this small, but I want to show a minimal structure in order to build up to something more maintainable, and also to show how simple setup.py can be. Here's a simple directory structure:

```
some_module_proj/
├── setup.py
└── some_module.py
```

The code we want to share is in some_module.py:

appendices/packaging/some_module_proj/some_module.py

```
def some_func():
return 42
```

To make it installable with pip, we need a setup.py file. This is about as bare bones as you can get:

appendices/packaging/some_module_proj/setup.py

```
from setuptools import setup


setup(
name='some_module',
py_modules=['some_module']
)
```

One directory with one module and a setup.py file is enough to make it installable via pip:

```
$ cd /path/to/code/appendices/packaging
$ pip install ./some_module_proj
Processing ./some_module_proj
Installing collected packages: some-module
Running setup.py install for some-module ... done
Successfully installed some-module-0.0.0
```

And we can now use some_module from Python (or from a test):

```
$ python
Python 3.6.1 (v3.6.1:69c0db5050, Mar 21 2017, 01:21:04)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from some_module import some_func
```

```
>>> some_func()
42
>>> exit()
```

That’s a minimal setup, but it’s not realistic. If you’re sharing code, odds are you are sharing a package. The next section builds on this to write a setup.py file for a package.

Creating an Installable Package

Let's make this code a package by adding an __init__.py and putting the __init__.py file and module in a directory with a package name:

```
$ tree some_package_proj/
some_package_proj/
├── setup.py
└── src
    └── some_package
        ├── __init__.py
        └── some_module.py
```

The content of some_module.py doesn't change. The __init__.py needs to be written to expose the module functionality to the outside world through the package namespace. There are lots of choices for this. I recommend skimming the two sections of the Python documentation[54] that cover this topic.

If we do something like this in __init__.py:

```
import some_package.some_module
```

the client code will have to specify some_module:

```
import some_package
some_package.some_module.some_func()
```

However, I'm thinking that some_module.py is really our API for the package, and we want everything in it to be exposed to the package level. Therefore, we'll use this form:

appendices/packaging/some_package_proj/src/some_package/__init__.py

```
from some_package.some_module import *
```

Now the client code can do this instead:

```
import some_package
some_package.some_func()
```

We also have to change the setup.py file, but not much:

appendices/packaging/some_package_proj/setup.py

```
from setuptools import setup, find_packages

setup(
name='some_package',
```

```
  packages=find_packages(where='src'),
  package_dir={'': 'src'},
  )
```

Instead of using py_modules, we specify packages.

This is now installable:

```
  $ cd /path/to/code/appendices/packaging
  $ pip install ./some_package_proj/
  Processing ./some_package_proj
  Installing collected packages: some-package
  Running setup.py install for some-package ... done
  Successfully installed some-package-0.0.0
```

and usable:

```
  $ python
  Python 3.6.1 (v3.6.1:69c0db5050, Mar 21 2017, 01:21:04)
  [GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
  Type "help", "copyright", "credits" or "license" for more information.
  >>> from some_package import some_func
  >>> some_func()
  42
```

Our project is now installable and in a structure that’s easy to build on. You can add a tests directory at the same level of src to add our tests if you want. However, the setup.py file is still missing some metadata needed to create a proper source distribution or wheel. It’s just a little bit more work to make that possible.

Creating a Source Distribution and Wheel

For personal use, the configuration shown in the previous section is enough to create a source distribution and a wheel. Let's try it:

```
$ cd /path/to/code/appendices/packaging/some_package_proj/
$ python setup.py sdist bdist_wheel
running sdist
...
warning: sdist: standard file not found:
should have one of README, README.rst, README.txt

running check
warning: check: missing required meta-data: url

warning: check: missing meta-data:
either (author and author_email)
or (maintainer and maintainer_email) must be supplied

running bdist_wheel
...
$ ls dist
some_package-0.0.0-py3-none-any.whl some_package-0.0.0.tar.gz
```

Well, with some warnings, a .whl and a .tar.gz file are created. Let's get rid of those warnings.

To do that, we need to:

- Add one of these files: README, README.rst, or README.txt.
- Add metadata for url.
- Add metadata for either (author and author_email) or (maintainer and maintainer_email).

Let's also add:

- A version number
- A license
- A change log

It makes sense that you'd want these things. Including some kind of README allows people to know how to use the package. The url, author, and author_email (or maintainer) information makes sense to let users know who to contact if they have issues or questions about the package. A license is important to let people know how they can distribute, contribute, and reuse the package. And if it's not open source, say so in the license data. To choose a license for open source projects, I recommend looking at https://choosealicense.com.

Those extra bits don’t add too much work. Here’s what I’ve come up with for a minimal default.

The setup.py:

appendices/packaging/some_package_proj_v2/setup.py

```
from setuptools import setup, find_packages

setup(
name='some_package',
description='Demonstrate packaging and distribution',

version='1.0',
author='Brian Okken',
author_email='brian@pythontesting.net',
url='https://pragprog.com/book/bopytest/python-testing-with-pytest',

packages=find_packages(where='src'),
package_dir={'': 'src'},
)
```

You should put the terms of the licensing in a LICENSE file. All of the code in this book follows the following license:

appendices/packaging/some_package_proj_v2/LICENSE

```
Copyright (c) 2017 The Pragmatic Programmers, LLC

All rights reserved.

Copyrights apply to this source code.

You may use the source code in your own projects, however the source code
may not be used to create commercial training material, courses, books,
articles, and the like. We make no guarantees that this source code is fit
for any purpose.
```

Here’s the README.rst:

appendices/packaging/some_package_proj_v2/README.rst

```
==================================================
some_package: Demonstrate packaging and distribution
==================================================

``some_package`` is the Python package to demostrate how easy it is
to create installable, maintainable, shareable packages and distributions.
```

```
It does contain one function, called ``some_func()``.

.. code-block

>>> import some_package
>>> some_package.some_func()
42


That's it, really.
```

The README.rst is formatted in reStructuredText.[55] I've done what many have done before me: I copied a README.rst from an open source project, removed everything I didn't like, and changed everything else to reflect this project.

You can also use an ASCII-formatted README.txt or README, but I'm okay with copy/paste/edit in this instance.

I recommend also adding a change log. Here's the start of one:

appendices/packaging/some_package_proj_v2/CHANGELOG.rst

```
Changelog
=========

----------------------------------------------------

1.0
---

Changes:
~~~~~~~~

- Initial version.
```

See http://keepachangelog.com for some great advice on what to put in your change log. All of the changes to tasks_proj over the course of this book have been logged into a CHANGELOG.rst file.

Let's see if this was enough to remove the warnings:

```
$ cd /path/to/code/appendices/packaging/some_package_proj_v2
$ python setup.py sdist bdist_wheel
running sdist
running build
```

```
running build_py
creating build
creating build/lib
creating build/lib/some_package
copying src/some_package/__init__.py
 -> build/lib/some_package
copying src/some_package/some_module.py
 -> build/lib/some_package
installing to build/bdist.macosx-10.6-intel/wheel
running install
running install_lib
creating build/bdist.macosx-10.6-intel
creating build/bdist.macosx-10.6-intel/wheel
creating build/bdist.macosx-10.6-intel/wheel/some_package
copying build/lib/some_package/__init__.py
 -> build/bdist.macosx-10.6-intel/wheel/some_package
copying build/lib/some_package/some_module.py
 -> build/bdist.macosx-10.6-intel/wheel/some_package
running install_egg_info
Copying src/some_package.egg-info to
build/bdist.macosx-10.6-intel/wheel/some_package-1.0-py3.6.egg-info
running install_scripts
creating build/bdist.macosx-10.6-intel/wheel/some_package-1.0.dist-info/WHEEL
```

```
$ ls dist
some_package-1.0-py3-none-any.whl some_package-1.0.tar.gz
```

Yep. No warnings.

Now, we can put the .whl and/or .tar.gz files in a local shared directory and pip install to our heart's content:

```
$ cd /path/to/code/appendices/packaging/some_package_proj_v2
$ mkdir ~/packages/
$ cp dist/some_package-1.0-py3-none-any.whl ~/packages
$ cp dist/some_package-1.0.tar.gz ~/packages
$ pip install --no-index --find-links=~/packages some_package
Collecting some_package
Installing collected packages: some-package
Successfully installed some-package-1.0
$ pip install --no-index --find-links=./dist some_package==1.0
Requirement already satisfied: some_package==1.0 in
/path/to/venv/lib/python3.6/site-packages
```

```
$
```

Now you can create your own stash of local project packages from your team, including multiple versions of each, and install them almost as easily as packages from PyPI.

Creating a PyPI-Installable Package

You need to add more metadata to your setup.py to get a package ready to distribute on PyPI. You also need to use a tool such as Twine[56] to push packages to PyPI. (Twine is a collection of utilities to help make interacting with PyPI easy and secure. It handles authentication over HTTPS to keep your PyPI credentials secure, and handles the uploading of packages to PyPI.)

This is now beyond the scope of this book. However, for information about how to start contributing through PyPI, take a look at the Python Packaging User Guide[57] and the the PyPI[58] section of the Python documentation.

Footnotes

[54] https://docs.python.org/3/tutorial/modules.html#packages

[55] http://docutils.sourceforge.net/rst.html

[56] https://pypi.python.org/pypi/twine

[57] https://python-packaging-user-guide.readthedocs.io

[58] https://docs.python.org/3/distutils/packageindex.html

Appendix 5
xUnit Fixtures

In addition to the fixture model described in Chapter 3, *pytest Fixtures*, pytest also supports xUnit style fixtures, which are similar to jUnit for Java, cppUnit for C++, and so on.

Generally, xUnit frameworks use a flow of control that looks something like this:

```
setup()
test_function()
teardown()
```

This is repeated for every test that will run. pytest fixtures can do anything you need this type of configuration for and more, but if you really want to have setup() and teardown() functions, pytest allows that, too, with some limitations.

Syntax of xUnit Fixtures

xUnit fixtures include setup()/teardown() functions for module, function, class, and method scope:

setup_module()/teardown_module()

> These run at the beginning and end of a module of tests. They run once each. The module parameter is optional.

setup_function()/teardown_function()

> These run before and after top-level test functions that are not methods of a test class. They run multiple times, once for every test function. The function parameter is optional.

setup_class()/teardown_class()

> These run before and after a class of tests. They run only once. The class parameter is optional.

setup_method()/teardown_method()

> These run before and after test methods that are part of a test class. They run multiple times, once for every test method. The method parameter is optional.

Here is an example of all the xUnit fixtures along with a few test functions:

appendices/xunit/test_xUnit_fixtures.py

```
  def setup_module(module):
  print(f'\nsetup_module() for {module.__name__}')


  def teardown_module(module):
  print(f'teardown_module() for {module.__name__}')


  def setup_function(function):
  print(f'setup_function() for {function.__name__}')


  def teardown_function(function):
  print(f'teardown_function() for {function.__name__}')


  def test_1():
  print('test_1()')
```

```
def test_2():
print('test_2()')


class TestClass:
@classmethod
def setup_class(cls):
print(f'setup_class() for class {cls.__name__}')

@classmethod
def teardown_class(cls):
print(f'teardown_class() for {cls.__name__}')

def setup_method(self, method):
print(f'setup_method() for {method.__name__}')

def teardown_method(self, method):
print(f'teardown_method() for {method.__name__}')

def test_3(self):
print('test_3()')

def test_4(self):
print('test_4()')
```

I used the parameters to the fixture functions to get the name of the module/function/class/method to pass to the print statement. You don't have to use the parameter names module, function, cls, and method, but that's the convention.

Here's the test session to help visualize the control flow:

```
$ cd /path/to/code/appendices/xunit
$ pytest -s test_xUnit_fixtures.py
============ test session starts =============
plugins: mock-1.6.0, cov-2.5.1
collected 4 items

test_xUnit_fixtures.py
setup_module() for test_xUnit_fixtures
setup_function() for test_1
test_1()
```

```
.teardown_function() for test_1
setup_function() for test_2
test_2()
.teardown_function() for test_2
setup_class() for class TestClass
setup_method() for test_3
test_3()
.teardown_method() for test_3
setup_method() for test_4
test_4()
.teardown_method() for test_4
teardown_class() for TestClass
teardown_module() for test_xUnit_fixtures

========== 4 passed in 0.01 seconds ==========
```

Mixing pytest Fixtures and xUnit Fixtures

You can mix pytest fixtures and xUnit fixtures:

appendices/xunit/test_mixed_fixtures.py

```
import pytest


def setup_module():
print('\nsetup_module() - xUnit')


def teardown_module():
print('teardown_module() - xUnit')


def setup_function():
print('setup_function() - xUnit')


def teardown_function():
print('teardown_function() - xUnit\n')
@pytest.fixture(scope='module')
def module_fixture():
print('module_fixture() setup - pytest')
yield
print('module_fixture() teardown - pytest')


@pytest.fixture(scope='function')
def function_fixture():
print('function_fixture() setup - pytest')
yield
print('function_fixture() teardown - pytest')


def test_1(module_fixture, function_fixture):
print('test_1()')


def test_2(module_fixture, function_fixture):
```

```
    print('test_2()')
```

You can do it. But please don't. It gets confusing. Take a look at this:

```
$ cd /path/to/code/appendices/xunit
$ pytest -s test_mixed_fixtures.py
============ test session starts =============
plugins: mock-1.6.0, cov-2.5.1
collected 2 items

test_mixed_fixtures.py
setup_module() - xUnit
setup_function() - xUnit
module_fixture() setup - pytest
function_fixture() setup - pytest
test_1()
.function_fixture() teardown - pytest
teardown_function() - xUnit

setup_function() - xUnit
function_fixture() setup - pytest
test_2()
.function_fixture() teardown - pytest
teardown_function() - xUnit

module_fixture() teardown - pytest
teardown_module() - xUnit


========== 2 passed in 0.01 seconds ==========
```

In this example, I've also shown that the module, function, and method parameters to the xUnit fixture functions are optional. I left them out of the function definition, and it still runs fine.

Limitations of xUnit Fixtures

Following are a few of the limitations of xUnit fixtures:

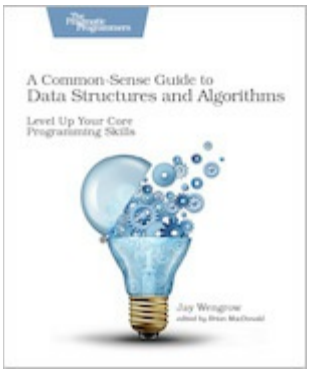
- xUnit fixtures don’t show up in -setup-show and -setup-plan. This might seem like a small thing, but when you start writing a bunch of fixtures and debugging them, you’ll love these flags.
- There are no session scope xUnit fixtures. The largest scope is module.
- Picking and choosing which fixtures a test needs is more difficult with xUnit fixtures. If a test is in a class that has fixtures defined, the test will use them, even if it doesn’t need to.
- Nesting is at most three levels: module, class, and method.
- The only way to optimize fixture usage is to create modules and classes with common fixture requirements for all the tests in them.
- No parametrization is supported at the fixture level. You can still use parametrized tests, but xUnit fixtures cannot be parametrized.

There are enough limitations of xUnit fixtures that I strongly encourage you to forget you even saw this appendix and stick with normal pytest fixtures.

You May Be Interested In…

Select a cover for more information

A Common-Sense Guide to Data Structures and Algorithms

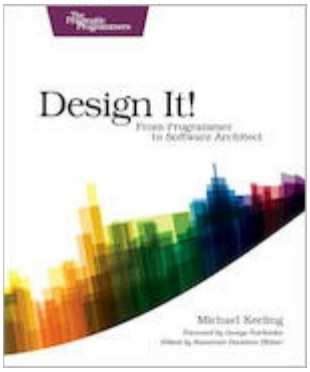


If you last saw algorithms in a university course or at a job interview, you're missing out on what they can do for your code. Learn different sorting and searching techniques, and when to use each. Find out how to use recursion effectively. Discover structures for specialized applications, such as trees and graphs. Use Big O notation to decide which algorithms are best for your production environment. Beginners will learn how to use these techniques from the start, and experienced developers will rediscover approaches they may have forgotten.

Jay Wengrow

(218 pages) ISBN: 9781680502442 $45.95

Design It!



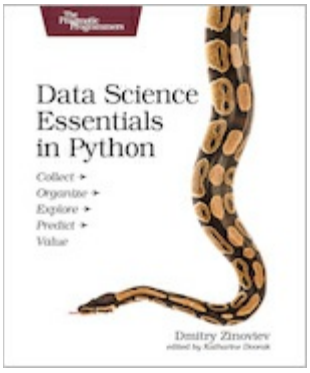
Don't engineer by coincidence—design it like you mean it! Grounded by fundamentals and filled with practical design methods, this is the perfect introduction to software architecture for programmers who are ready to grow their design skills. Ask the right stakeholders the right questions, explore design options, share your design decisions, and facilitate collaborative workshops that are fast, effective, and fun. Become a better programmer, leader, and designer. Use your new skills to lead your team in implementing software with the right capabilities—and

develop awesome software!

Michael Keeling

(350 pages) ISBN: 9781680502091 $42.50

Data Science Essentials in Python

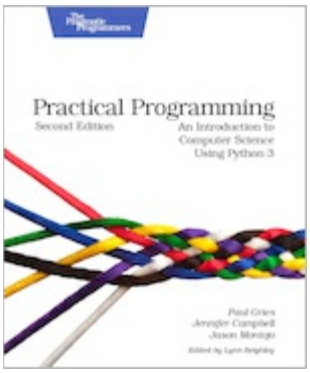


Go from messy, unstructured artifacts stored in SQL and NoSQL databases to a neat, well-organized dataset with this quick reference for the busy data scientist. Understand text mining, machine learning, and network analysis; process numeric data with the NumPy and Pandas modules; describe and analyze data using statistical and network-theoretical methods; and see actual examples of data analysis at work. This one-stop solution covers the essential data science you need in Python.

Dmitry Zinoviev

(224 pages) ISBN: 9781680501841 $29

Practical Programming (2nd edition)



This book is for anyone who wants to understand computer programming. You'll learn to program in a language that's used in millions of smartphones, tablets, and PCs. You'll code along with the book, writing programs to solve real-world problems as you learn the fundamentals of programming using Python 3. You'll learn about design, algorithms, testing, and debugging, and come away with all the tools you need to produce quality code. In this second edition, we've updated almost all the material, incorporating the lessons we've learned over the past five years of teaching Python to people new to programming.

Paul Gries, Jennifer Campbell, Jason Montojo

(400 pages) ISBN: 9781937785451 $38

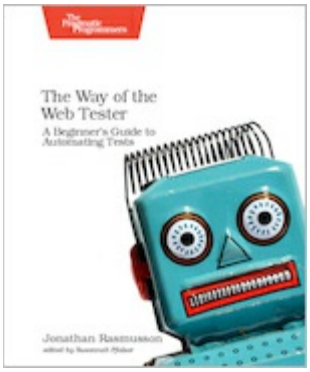
Explore It!

Uncover surprises, risks, and potentially serious bugs with exploratory testing. Rather than designing all tests in advance, explorers design and execute small, rapid experiments, using what they learned from the last little experiment to inform the next. Learn essential skills of a master explorer, including how to analyze software to discover key points of vulnerability, how to design experiments on the fly, how to hone your observation skills, and how to focus your efforts.

Elisabeth Hendrickson

(186 pages) ISBN: 9781937785024 $29

The Way of the Web Tester



This book is for everyone who needs to test the web. As a tester, you'll automate your tests. As a developer, you'll build more robust solutions. And as a team, you'll gain a vocabulary and a means to coordinate how to write and organize automated tests for the web. Follow the testing pyramid and level up your skills in user interface testing, integration testing, and unit testing. Your new skills will free you up to do other, more important things while letting the computer do the one thing it's really good at: quickly running thousands of repetitive tasks.

Jonathan Rasmusson

(256 pages) ISBN: 9781680501834 $29

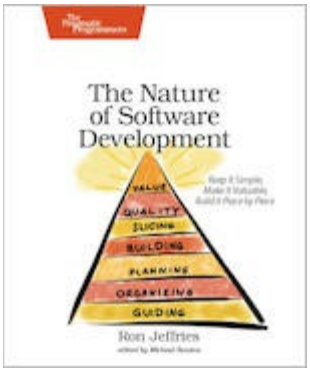
Your Code as a Crime Scene

Jack the Ripper and legacy codebases have more in common than you'd think. Inspired by forensic psychology methods, this book teaches you strategies to predict the future of your codebase, assess refactoring direction, and understand how your team influences the design. With its unique blend of forensic psychology and code analysis, this book arms you with the strategies you need, no matter what programming language you use.

Adam Tornhill

(218 pages) ISBN: 9781680500387 $36

The Nature of Software Development

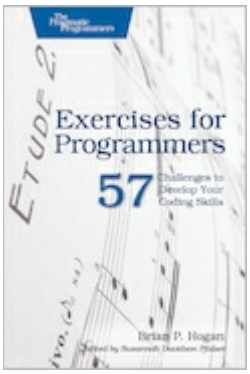


You need to get value from your software project. You need it "free, now, and perfect." We can't get you there, but we can help you get to "cheaper, sooner, and better." This book leads you from the desire for value down to the specific activities that help good Agile projects deliver better software sooner, and at a lower cost. Using simple sketches and a few words, the author invites you to follow his path of learning and understanding from a half century of software development and from his engagement with Agile methods from their very beginning.

Ron Jeffries

(176 pages) ISBN: 9781941222379 $24

Exercises for Programmers



When you write software, you need to be at the top of your game. Great programmers practice to keep their skills sharp. Get sharp and stay sharp with more than fifty practice exercises rooted in real-world scenarios. If you're a new programmer, these challenges will help you learn what you need to break into the field, and if you're a seasoned pro, you can use these exercises to learn that hot new language for your next gig.

Brian P. Hogan

(118 pages) ISBN: 9781680501223 $24

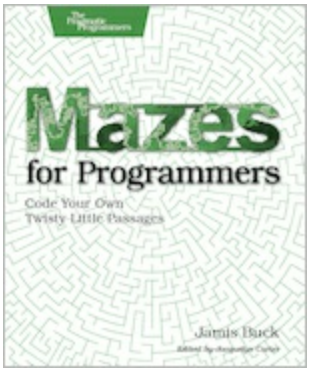
Creating Great Teams

People are happiest and most productive if they can choose what they work on and who they work with. Self-selecting teams give people that choice. Build well-designed and efficient teams to get the most out of your organization, with step-by-step instructions on how to set up teams quickly and efficiently. You'll create a process that works for you, whether you need to form teams from scratch, improve the design of existing teams, or are on the verge of a big team re-shuffle.

Sandy Mamoli and David Mole

(102 pages) ISBN: 9781680501285 $17

Mazes for Programmers

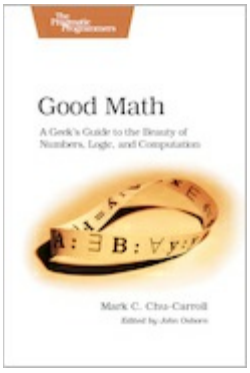


A book on mazes? Seriously? Yes! Not because you spend your day creating mazes, or because you particularly like solving mazes. But because it’s fun. Remember when programming used to be fun? This book takes you back to those days when you were starting to program, and you wanted to make your code do things, draw things, and solve puzzles. It’s fun because it lets you explore and grow your code, and reminds you how it feels to just think. Sometimes it feels like you live your life in a maze of twisty little passages, all alike. Now you can code your way out.

Jamis Buck

(286 pages) ISBN: 9781680500554 $38

Good Math



Mathematics is beautiful—and it can be fun and exciting as well as practical. Good Math is your guide to some of the most intriguing topics from two thousand years of mathematics: from Egyptian fractions to Turing machines; from the real meaning of numbers to proof trees, group symmetry, and mechanical computation. If you’ve ever wondered what lay beyond the proofs you struggled to complete in high school geometry, or what limits the capabilities of the computer on your desk, this is the book for you.

Mark C. Chu-Carroll

(282 pages) ISBN: 9781937785338 $34