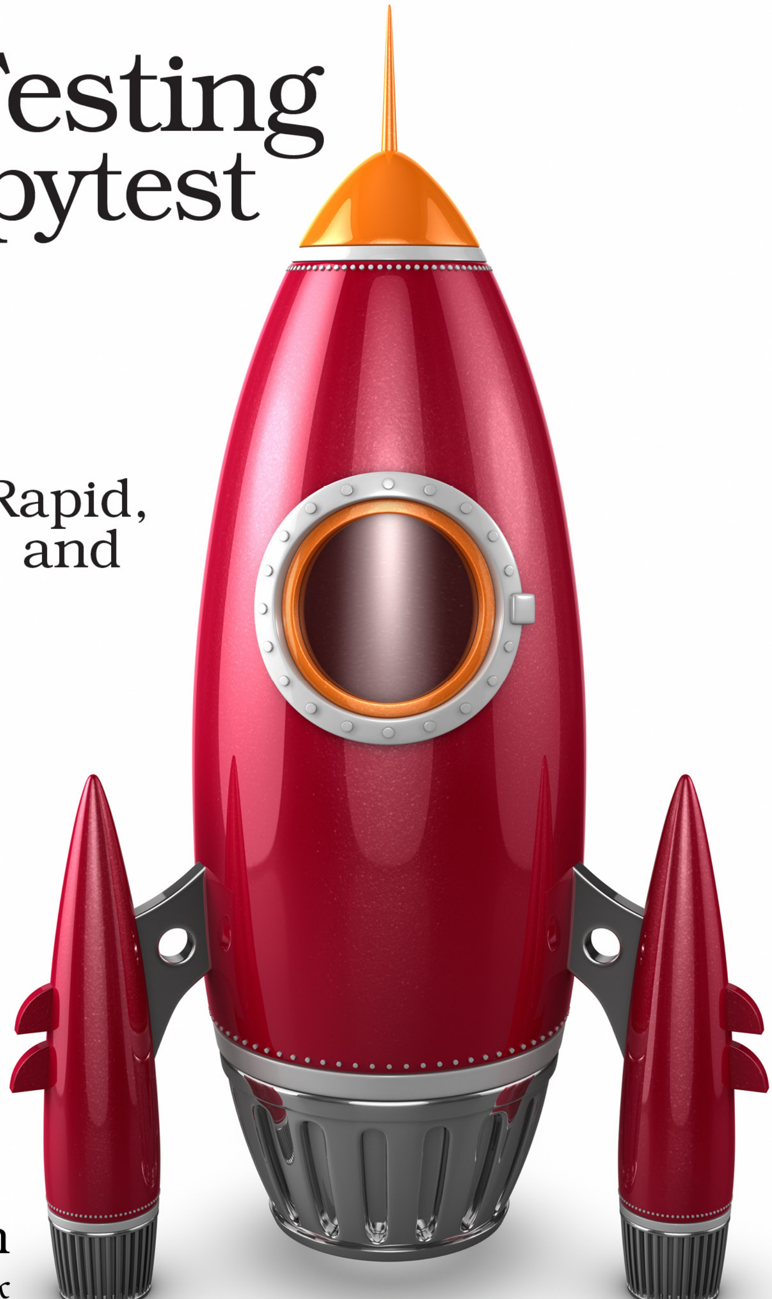
The Pragmatic Programmers

Python Testing with pytest

Simple, Rapid, Effective, and Scalable



Brian Okken

edited by Katharine Dvorak

Under Construction: The book you're reading is still under development. As part of our Beta book program, we're releasing this copy well before a normal book would be released. That way you're able to get this content a couple of months before it's available in finished form, and we'll get feedback to make the book even better. The idea is that everyone wins!

Be warned: The book has not had a full technical edit, so it will contain errors. It has not been copyedited, so it will be full of typos, spelling mistakes, and the occasional creative piece of grammar. And there's been no effort spent doing layout, so you'll find bad page breaks, over-long code lines, incorrect hyphenation, and all the other ugly things that you wouldn't expect to see in a finished book. It also doesn't have an index. We can't be held liable if you use this book to try to create a spiffy application and you somehow end up with a strangely shaped farm implement instead. Despite all this, we think you'll enjoy it!

Download Updates: Throughout this process you'll be able to get updated ebooks from your account at *pragprog.com/my_account*. When the book is complete, you'll get the final version (and subsequent updates) from the same address.

Send us your feedback: In the meantime, we'd appreciate you sending us your feedback on this book at *pragprog.com/titles/bopytest/errata*, or by using the links at the bottom of each page.

Thank you for being part of the Pragmatic community!

Andy

Python Testing with pytest

Simple, Rapid, Effective, and Scalable

Brian Okken

The Pragmatic Bookshelf
Raleigh, North Carolina

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

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

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

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

Printed in the United States of America.
ISBN-13: 978-1-68050-240-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: B2.0—June 7, 2017

Contents

Change History

The book you're reading is in beta. This means we update it frequently. This page lists the major changes that have been made at each beta release of the book, with the most recent changes first.

Beta 2.0—June 7, 2017

- We added Chapter 6, *Configuration*, on page 107, which covers configuration files and settings.
- We changed the code sample so that only one application—code/tasks_proj—needs to be installed for Chapters 1 through 6. The modifications in code/ch1 through code/ch6 are just to the test code.
- All current errata have been addressed.

Beta 1.0—May 24, 2017

- Initial beta release.

Preface

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

Enter pytest.

What Is pytest?

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

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

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

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

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

Learn pytest While Testing an Example Application

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

The Tasks Project

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

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

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

Here's an example session:

```
$ tasks add 'do something' --owner Brian
$ tasks add 'do something else'
$ tasks list
```

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

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

Test Strategy

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

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

- *Subcutaneous test:* A test that doesn't run against the final end-user interface, but against an interface just below the surface. Since most of the tests in this book test against the API layer—not the CLI—they qualify as subcutaneous tests.

How This Book Is Organized

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

In Chapter 2, *Writing Test Functions*, on page 21, you'll install Tasks locally using pip and look at how to structure tests within a Python project. We'll do this so that we can get to writing tests against a real application. All the examples in this chapter run tests against the installed application, including writing to the database. The actual test functions are the focus of this chapter, and you'll learn how to use assert 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 we'll cover how to run just some of the tests, not just with markers, but by structuring our test code into directories, modules, and classes, and how to run these subsets of tests.

Not all of your test code goes into test functions. In Chapter 3, *pytest Fixtures*, on page 47, you'll learn how to put test data into test fixtures, as well as setup and teardown code. Setting up system state (or subsystem or unit state) is an important part of software testing. We explore this aspect of pytest fixtures to help us with the Tasks project in getting the database initialized and pre-filling it with test data for some tests. Fixtures are an incredibly powerful part of pytest, and you'll learn how to use them effectively to further reduce test code duplication and help make your test code incredibly readable and maintainable. pytest fixtures are also parameterizable, similar to test functions, and we use this feature to be able to run all of our tests against both TinyDB and MongoDB, the database backends supported by Tasks.

In Chapter 4, *Builtin Fixtures*, on page 69, we'll look at some builtin fixtures provided out-of-the-box by pytest. You'll learn how pytest builtin fixtures can keep track of temporary directories and files for you, can help you test output from your code under test, use monkey patches, check for warnings, and more.

In Chapter 5, *Plugins*, on page 91, you'll learn how to add command-line options to pytest, alter the pytest output, and share pytest customizations, including fixtures, with others through writing, packaging, and distributing your own plugins. The plugin we develop in this chapter is used to make the test failures we see while testing Tasks just a little bit nicer. We also look at how to properly test our test plugins. How's that for meta? And just in case you're not inspired enough by this chapter to write some plugins of your own, I've hand-picked a bunch of great plugins to show off what's possible in Appendix 3, *Plugin Sampler Pack*, on page 123.

Speaking of customization, in Chapter 6, *Configuration*, on page 107, you'll learn how you can customize how pytest runs by default for your project with configuration files. With a pytest.ini file, you can do things like store command-line options so you don't have to type them all the time, tell pytest to not look into certain directories for test files, specify a minimum pytest version your tests are written for, and more. These configuration elements can be put in tox.ini or setup.cfg as well.

In the final chapter, Chapter 7, *Using pytest with Other Tools*, on page 117, we'll look at how we can take the already powerful pytest and supercharge our testing with some complementary tools. You'll run the Tasks project on multiple versions of Python with tox. You'll test the Tasks CLI while not having to run the rest of the system with mock. You'll use *coverage* to see if any of the Tasks source code isn't being tested. You'll use pytest to run legacy unittest tests, and check the docstring code examples with doctest.

What You Need To Know

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

pip
: You should use pip to install pytest and pytest plugins. If you want a refresher on pip, check out Appendix 2, *pip*, on page 121.

A command line
: I wrote this book and captured the example output using bash on a Mac laptop. However, the only commands I use in bash are `cd` to go to a specific directory, and `pytest`, of course. Since `cd` exists in Windows `cmd.exe` and all

unix shells that I know of, all examples should be runnable on whatever terminal-like application you choose to use.

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

Example Code and Online Resources

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

I've been programming for over 25 years, and nothing has made me love writing test code as much as pytest. I hope you learn a lot from this book, and I hope that you'll end up loving test code as much as I do.

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

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 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 we need more information, we can use -v or --verbose:

```
$ pytest -v test_one.py
===================== test session starts ======================
cachedir: .cache
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 it's own section to show us why it failed. And pytest tells us exactly where 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 ======================
cachedir: .cache
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, we'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'll not be using overly silly tests like this in the rest of the book. We'll be looking 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 built-in 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, we'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 are using for testing:

```
$ mkdir sandbox
$ cd sandbox
$ python3 -m virtualenv venv
$ source venv/bin/activate
$ pip install pytest
```

If you aren't familiar with virtualenv or pip, we've got you covered. Check out Appendix 1, *Virtual Environments*, on page 119 and Appendix 2, *pip*, on page 121.

What about Windows, Python 2, and venv?

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

On Windows, the '$' prompt is 'C:\path\>'. Other than that, the only thing that needs to change is the line where you activate the virtual environment. Do this:

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

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 file name, 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

```python
from collections import namedtuple

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


def test_defaults():
    t1 = Task()
    t2 = Task(None, None, False, None)
    assert t1 == t2

def test_member_access():
    t = Task('buy milk', 'brian')
    assert t.summary == 'buy milk'
    assert t.owner == 'brian'
    assert (t.done, t.id) == (False, None)
```

I want to be able to create Task objects without having to specify all the fields. So defaults would be cool. I recently read about using __new__.__defaults__, so test_defaults() is there to validate my understanding of how the defaults work.

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

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

ch1/tasks/test_four.py

```python
from collections import namedtuple

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


def test_asdict():
    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():
    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 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 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*, on page 107.

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

```
$ pytest test_three.py
===================== test session starts ======================
platform darwin -- Python 3.5.3, pytest-3.0.7, py-1.4.33, pluggy-0.4.0
rootdir: /path/to/code/ch1/tasks, inifile:
collected 2 items

test_three.py ..

=================== 2 passed in 0.02 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 we talk about session scope in relation to pytest fixtures in *Specifying Fixture Scope*, on page 54.

platform darwin -- Python 3.5.3, pytest-3.0.7, py-1.4.33, 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`. We look at configuration files in more detail in Chapter 6, *Configuration*, on page 107.

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 0.02 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 is 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*, on page 33
- 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*, on page 36
- 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*, on page 47, or in a hook function, discussed in Chapter 5, *Plugins*, on page 91.

Running Only One Test

One of the first things you'll want to do once you have started writing tests is to be able to run just one. I bring this up now because I don't want you to try to rely on the -k EXPRESSION option to run a single test. There is an easier way.

Specify the file directly, and add a ::test_name, like this:

```
$ pytest -v tasks/test_four.py::test_asdict
=================== test session starts ===================
```

```
cachedir: .cache
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.

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 are 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 : report which tests will be run

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 we start in the "ch1" directory, we should see all of the test functions we've looked at so far in this chapter:

```
$ 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 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:

```
$ 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 ===================
cachedir: .cache
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:

```
$ pytest -v -m run_these_please
================== test session starts ===================
cachedir: .cache
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. We'll discuss markers more completely in *Marking Test Functions*, on page 30.

-x, –exitfirst

Normal pytest behavior is to run every test it finds. If a test function encounters an 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 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:

```
$ 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 we see that all six tests (or "items") were collected, and in the bottom line we 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 we've already seen it and don't need to see it again:

```
$ 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.

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:

```
$ 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 0.24 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=now 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 filedescriptors 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:

```
$ 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:

```
$ 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 it's 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:

```
$ pytest -v --ff --tb=no
=================== test session starts ===================
cachedir: .cache
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:

```
$ 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:

```
$ 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 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:

```
$ 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.

–durations=N

The --durations=N option is incredibly helpful when you are 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:

```
$ 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. We'll cover fixtures in depth in Chapter 3, *pytest Fixtures*, on page 47.

–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, 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*, on page 107
- a list of environmental variables that can affect pytest behavior (also discussed in Chapter 6, *Configuration*, on page 107)
- a reminder that pytest --markers can be used to see available markers, discussed in Chapter 2, *Writing Test Functions*, on page 21
- a reminder that pytest --fixtures can be used to see available fixtures, discussed in Chapter 3, *pytest Fixtures*, on page 47

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 are running. This is because along the path to a specified file or directory, pytest may find config.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 config.py files and test files allows customized behavior local to a project or even a subset of the tests for a project. We'll learn about config.py and ini files such as pytest.ini in Chapter 6, *Configuration*, on page 107.

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 are 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*, on page 119 if you are 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*, on page 121 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 look at writing test functions, parameterizing 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 are 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, we'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 we'll look at using assert in tests, how tests handle unexpected exceptions, and testing for expected exceptions.

Eventually we'll have a lot of tests. Therefore, we'll look at how we can organize tests into classes, modules, and directories. We'll then use markers to mark which tests we want to run and discuss how builtin markers can help us skip tests and mark tests as expecting to fail. Finally, we'll also cover parameterizing 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*, on page x, 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*, on page 125 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/
├── 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
    │   ├── test_delete.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, LICENSE, README.rst, MANIFEST.in, and setup.py are discussed in more detail in Appendix 4, *Packaging and Distributing Python Projects*, on page 125. Although the 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. We'll learn all about pytest.ini in Chapter 6, *Configuration*, on page 107.

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*, on page 47 and Chapter 4, *Builtin Fixtures*, on page 69, and hook functions are discussed in Chapter 5, *Plugins*, on page 91.) 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.[1] 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*, on page 4, 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

```
from tasks import Task
```

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

```
def test_asdict():
    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_from_dict():
    t_dict = {'summary': 'do something',
              'owner': 'okken',
              'done': True,
              'id': 2}
    t_task = Task(**t_dict)
    expected = Task('do something', 'okken', True, 2)
    assert t_task == expected

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, 10)
    assert t_after == t_expected

def test_defaults():
    t1 = Task()
    t2 = Task(None, None, False, None)
    assert t1 == t2

def test_member_access():
    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 tasks_proj
$ pip install .
Processing /path/to/code/tasks_proj
Collecting click (from tasks==0.1.0)
  Using cached click-6.7-py2.py3-none-any.whl
Collecting tinydb (from tasks==0.1.0)
  Downloading tinydb-3.2.3.tar.gz
```

```
Installing collected packages: click, tinydb, tasks
  Running setup.py install for tinydb ... done
  Running setup.py install for tasks ... done
Successfully installed click-6.7 tasks-0.1.0 tinydb-3.2.3
```

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 .
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)
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 our test:

```
$ cd tests/unit
$ pytest test_Task.py
===================== test session starts =====================
collected 5 items

test_Task.py .....

================== 5 passed in 0.03 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 have used any other testing framework, you've probably seen various assert helper functions. For example, following is a list of a few of the assert forms and assert helper functions:

pytest	unittest
assert something	assertTrue(something)

pytest	unittest
`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 fails. Boolean operators like `a == b` and `a < b` are obvious expressions that have boolean value, but lots of other things do as well. In fact, you can evaluate every expression in Python in a boolean context. Here are a few examples:

```
>>> bool(False)
False
>>> bool(None)
False
>>> bool(0)
False
>>> bool(0.0)
False
>>> bool(0j) # this is for imaginary numbers
False
>>> bool('') # empty string
False
>>> bool(()) # empty tuple
False
>>> bool([]) # empty list
False
>>> bool({}) # empty dictionary
False
>>> bool(True)
True
>>> bool(1)
True
>>> bool(0.2)
True
>>> bool(3j)
True
>>> bool('hi') # non-empty string
True
>>> bool((1,2)) # non-empty tuple
True
>>> bool(['a','b']) # non-empty list
True
>>> bool({'c':2}) # non-empty dictionary
True
```

If something is a number type, such as an `int` or `float` that has a zero value, it evaluates to `False`. For container types, if the `len()` of the object is 0, then the boolean value is False. Container objects with a non-zero length evaluate to True. Therefore, you can put pretty much any expression in an assert statement, if it makes sense. If the expression in assert <expression> evaluates to False, your test fails.

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

```
from tasks import Task

def test_task_equality():
    t1 = Task('sit there', 'brian')
    t2 = Task('do something', 'okken')
    assert t1 == t2

def test_dict_equality():
    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:

```
$ 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 =====================
cachedir: ../.cache
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 also it 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.[2]

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

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(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 is 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_TypeError():
    with pytest.raises(TypeError):
        tasks.add(task='not a Task object')
```

In test_add_TypeError() 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_TypeError(). We 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'. We 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_ValueError():
    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. Markers are like labels, or tags in social media. 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_TypeError():
    with pytest.raises(TypeError):
        tasks.list(owner=123)

@pytest.mark.get
@pytest.mark.smoke
def test_get_TypeError():
    with pytest.raises(TypeError):
        tasks.get(task_id='123')
```

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

```
$ pytest -v -m 'smoke' test_api_exceptions.py
===================== test session starts =====================
cachedir: ../.cache
collected 8 items

test_api_exceptions.py::test_list_TypeError PASSED
test_api_exceptions.py::test_get_TypeError PASSED

===================== 6 tests deselected ======================
$ pytest -v -m 'get' test_api_exceptions.py
===================== test session starts =====================
cachedir: ../.cache
```

```
collected 8 items

test_api_exceptions.py::test_get_TypeError PASSED

==================== 7 tests deselected =====================
=========== 1 passed, 7 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' ran both tests marked with @pytest.mark.smoke. Using -m 'get' ran 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 ====================
cachedir: ../.cache
collected 8 items

test_api_exceptions.py::test_get_TypeError PASSED

==================== 7 tests deselected =====================
=========== 1 passed, 7 deselected in 0.01 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 ====================
cachedir: ../.cache
collected 8 items

test_api_exceptions.py::test_list_TypeError PASSED

==================== 7 tests deselected =====================
=========== 1 passed, 7 deselected in 0.01 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():
    # GIVEN an initialized tasks db
    # WHEN a new task is added
    new_task = Task('do something')
    task_id = tasks.add(new_task)
    # THEN returned task_id is of type int
    assert isinstance(task_id, int)

@pytest.mark.smoke
def test_added_task_has_id_set():
    # 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
    # AND contents are equivalent (except for id)
    # the [:-1] syntax retruns a list with all but the last element
    assert task_from_db[:-1] == new_task[:-1]
```

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):
    # 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*, on page 69, and you'll learn about writing your own fixtures and how they work in Chapter 3, *pytest Fixtures*, on page 47, 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. We'll look at all that and more in later chapters, but we needed 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*, on page 127.)

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

```
$ cd ch2/tasks_proj/
$ pytest -v -m smoke
===================== test session starts =====================
cachedir: .cache
collected 16 items

tests/func/test_add.py::test_added_task_has_id_set PASSED
tests/func/test_api_exceptions.py::test_list_TypeError PASSED
tests/func/test_api_exceptions.py::test_get_TypeError PASSED

===================== 13 tests deselected =====================
=========== 3 passed, 13 deselected in 0.07 seconds ===========
```

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

Skipping Tests

While the markers discussed in *Marking Test Functions*, on page 30 were names of your own choosing, pytest includes a few helpful builtin markers: skip, skipif, and xfail. We'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
from tasks import Task

def test_unique_id():
    id_1 = tasks.unique_id()
    id_2 = tasks.unique_id()
    assert id_1 != id_2
```

Then give it a run:

```
$ pytest test_unique_id_1.py
=========================== test session starts ===========================
collected 1 items
```

```
test_unique_id_1.py F

================================= FAILURES =================================
______________________________ test_unique_id ______________________________

    def test_unique_id():
        id_1 = tasks.unique_id()
        id_2 = tasks.unique_id()
>       assert id_1 != id_2
E       assert 1 != 1

test_unique_id_1.py:9: AssertionError
======================== 1 failed in 0.05 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

```python
@pytest.mark.skip(reason='misunderstood the API')
def test_unique_id_1():
    id_1 = tasks.unique_id()
    id_2 = tasks.unique_id()
    assert id_1 != id_2

def test_unique_id_2():
    # add a few things
    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 ===========================
cachedir: ../.cache
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():
    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 are 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 ===========================
cachedir: ../.cache
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():
    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():
    uid = tasks.unique_id()
    assert uid == 'a duck'

@pytest.mark.xfail()
def test_unique_id_not_a_duck():
    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:

```
$ 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 ========================
cachedir: ../.cache
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 an pytest.ini:

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

We'll discuss pytest.ini more in Chapter 6, *Configuration*, on page 107.

Running a Subset of Tests

We'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. We haven't seen test classes used yet, so we'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:

```
$ pytest tests/func
=========================== test session starts ===========================
collected 9 items

tests/func/test_add.py ..
tests/func/test_api_exceptions.py .......

======================== 9 passed in 0.06 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
=========================== test session starts ===========================
cachedir: tests/.cache
collected 9 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_TypeError PASSED
tests/func/test_api_exceptions.py::test_list_TypeError PASSED
tests/func/test_api_exceptions.py::test_get_TypeError PASSED
tests/func/test_api_exceptions.py::TestUpdate::test_bad_id PASSED
tests/func/test_api_exceptions.py::TestUpdate::test_bad_task PASSED
tests/func/test_api_exceptions.py::test_delete_TypeError PASSED
tests/func/test_api_exceptions.py::test_start_tasks_db_ValueError PASSED

======================== 9 passed in 0.05 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:

```
$ 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:

```
$ pytest -v tests/func/test_add.py::test_add_returns_valid_id
=========================== test session starts ============================
cachedir: tests/.cache
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():

    def test_bad_id(self):
        with pytest.raises(TypeError):
```

```
            tasks.update(task_id={'dict instead':1},
                         task=tasks.Task())

    def test_bad_task(self):
        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:

```
$ pytest -v tests/func/test_api_exceptions.py::TestUpdate
=========================== test session starts ===========================
cachedir: tests/.cache
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 tests
$ pytest -v func/test_api_exceptions.py::TestUpdate::test_bad_id
=========================== test session starts ===========================
cachedir: .cache
collected 8 items

func/test_api_exceptions.py::TestUpdate::test_bad_id PASSED
======================== 1 passed in 0.01 seconds =========================
```

The only reason I added cd tests first was to shorten the command line for the book.

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 don'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 test_add in their name:

```
$ pytest -v -k test_add
=========================== test session starts ===========================
cachedir: .cache
collected 16 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_TypeError PASSED

=========================== 13 tests deselected ===========================
================= 3 passed, 13 deselected in 0.06 seconds =================
```

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

```
$ pytest -v -k 'test_add and not has_id'
=========================== test session starts ===========================
cachedir: .cache
collected 16 items

tests/func/test_add.py::test_add_returns_valid_id PASSED
tests/func/test_api_exceptions.py::test_add_TypeError PASSED

=========================== 14 tests deselected ===========================
================= 2 passed, 14 deselected in 0.05 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.

Parameterized 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. Parameterized 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 parameterized 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():
    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):
    # someTask[:-1] -> a tuple containing (summary, owner, done)
    # i.e. compare everything but the id field
    return t1[:-1] == t2[:-1]

@pytest.fixture(autouse=True)
def initialized_tasks_db(tmpdir):
    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:

```
$ pytest -v  test_add_variety.py::test_add_1
======================== test session starts ========================
cachedir: ../.cache
collected 15 items

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 trough 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('excercise', 'BrIaN', False)])
def test_add_2(task):
```

```
    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 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:

```
$ pytest -v  test_add_variety.py::test_add_2
======================== test session starts ========================
cachedir: ../.cache
collected 21 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 0.06 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):
    task = Task(summary, owner, done)
    task_id = tasks.add(task)
    t_from_db = tasks.get(task_id)
    assert equivalent(t_from_db, task)
```

When we 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:

```
$ pytest -v  test_add_variety.py::test_add_3
======================== test session starts ========================
cachedir: ../.cache
collected 21 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 0.06 seconds ======================
```

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

```
$ pytest -v test_add_variety.py::test_add_3[sleep-None-False]
======================== test session starts ========================
cachedir: ../.cache
collected 18 items

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

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

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

```
$ pytest -v 'test_add_variety.py::test_add_3[eat eggs-BrIaN-False]'
======================== test session starts ========================
cachedir: ../.cache
collected 18 items

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

===================== 1 passed in 0.04 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('breathe', 'BRIAN', True),
                Task('excercise', 'BrIaN', False))

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

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 us 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):
    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:

```
======================== test session starts ========================
cachedir: ../.cache
collected 21 items

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

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

And these test identifiers can be used to run tests:

```
$ pytest -v 'test_add_variety.py::test_add_5[Task(wake,brian,False)]'
======================== test session starts ========================
cachedir: ../.cache
collected 18 items

test_add_variety.py::test_add_5[Task(wake,brian,False)] PASSED

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

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

We can apply parametrize() to classes as well. When we 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():
    def test_equivilent(self, task):
        task_id = tasks.add(task)
        t_from_db = tasks.get(task_id)
        assert equivalent(t_from_db, task)

    def test_valid_id(self, task):
        task_id = tasks.add(task)
        t_from_db = tasks.get(task_id)
        assert t_from_db.id == task_id
```

Here it is in action:

```
$ pytest -v test_add_variety.py::TestAdd
======================== test session starts ========================
cachedir: ../.cache
collected 25 items

test_add_variety.py::TestAdd::test_equivalent[Task(sleep,None,True)] PASSED
test_add_variety.py::TestAdd::test_equivalent[Task(wake,brian,False)] PASSED
test_add_variety.py::TestAdd::test_equivalent[Task(breathe,BRIAN,True)] PASSED
```

```
test_add_variety.py::TestAdd::test_equivalent[Task(excercise,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)] PASSED
test_add_variety.py::TestAdd::test_valid_id[Task(breathe,BRIAN,True)] PASSED
test_add_variety.py::TestAdd::test_valid_id[Task(excercise,BrIaN,False)] PASSED

===================== 8 passed in 0.06 seconds ======================
```

Exercises

1. Download the project for this chapter, tasks_proj, from the book's webpage[3] 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

We've run through a lot of the power of pytest in this chapter. Even with just what we've covered here, you can start supercharging your test suites. In many of the examples we 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 will take a deep dive into the wonderful world of pytest fixtures.

3. 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 42

def test_some_data(some_data):
    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), we need to address that 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*, on page 91. 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 our tests in the Tasks project will want to 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*, on page 69.)

So, given those pieces, this fixture works nicely:

ch3/a/tasks_proj/tests/conftest.py
```
import pytest
import tasks

@pytest.fixture()
def tasks_db(tmpdir):
    # 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." 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):
```

```
    # GIVEN an initialized tasks db
    # WHEN a new task is added
    new_task = Task('do something')
    task_id = tasks.add(new_task)
    # THEN returned task_id is of type int
    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.

Tracing Fixture Execution with –setup-show

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

```
$ cd ch3/tasks_proj/tests/func
$ pytest -v test_add.py -k valid_id
======================== test session starts ========================
cachedir: ../.cache
collected 2 items

test_add.py::test_add_returns_valid_id PASSED

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

When I'm developing fixtures, I like to see what's running and when. 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 2 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

======================== 1 tests deselected =========================
============== 1 passed, 1 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, we 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. We'll talk about scope in *Specifying Fixture Scope*, on page 54.

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

def test_a_tuple(a_tuple):
    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:

```
$ pytest  test_fixtures.py::test_a_tuple
======================== test session starts ========================
collected 3 items

test_fixtures.py F

============================= FAILURES ==============================
___________________________ test_a_tuple ____________________________

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

    def test_a_tuple(a_tuple):
>       assert a_tuple[3]['bar'] == 32
E       assert 23 == 32

test_fixtures.py:19: AssertionError
===================== 1 failed in 0.04 seconds ======================
```

Along with the stack trace section, pytest reports the value of the fixtures of failing tests.

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

```
$ pytest -v test_fixtures.py::test_other_data
======================== test session starts ========================
cachedir: .cache
collected 4 items
```

```
test_fixtures.py::test_other_data ERROR

============================== ERRORS ===============================
_________________ ERROR at setup of test_other_data _________________

    @pytest.fixture
    def some_other_data():
        x = 43
>       assert x == 42
E       assert 43 == 42

test_fixtures.py:16: AssertionError
====================== 1 error in 0.03 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

```python
from tasks import Task

# Reminder of Task constructor interface
# Task(summary=None, owner=None, done=False, id=None)
# Don't set id, it's set by database
# owner and done are optional

@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', 'Anna', 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('Teach people', 'Carrie'),
        Task('Make some videos', 'Carrie'),
        Task('Inspire', 'Carrie'),

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

```
        )
```

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

Using Multiple Fixtures

We've already seen that tmpdir uses tmpdir_factory. And we 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):
    'tasks 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):
    'tasks db 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):
    # 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 are actually testing, not 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:

```
$ pytest --setup-show test_add.py::test_add_increases_count
======================== test session starts ========================
collected 4 items

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 0.02 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

```
import pytest

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

@pytest.fixture(scope='module')
def mod_scope():
    pass

@pytest.fixture(scope='session')
def sess_scope():
    pass

def test_1(sess_scope, mod_scope, func_scope):
    pass

def test_2(sess_scope, mod_scope, func_scope):
    pass


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

@pytest.mark.usefixtures('class_scope')
class TestSomething():

    def test_3(self):
        pass

    def test_4(self):
        pass
```

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

```
$ 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 we 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 do not 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 the Task project fixtures.

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.

In order to have something like tasks_db be session scope, we need to use tmpdir_factory, since tmpdir is fixture 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

```python
import pytest
import tasks

@pytest.fixture(scope='session')
def tasks_db_session(tmpdir_factory):
    'initialize tasks db'
    # Setup : start db
```

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

    yield # this is where the testing happens

    # Teardown : stop db
    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 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.

However, 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

```
from tasks import Task

# Reminder of Task constructor interface
# Task(summary=None, owner=None, done=False, id=None)
# Don't set id, it's set by database
# owner and done are optional

@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', 'Anna', 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('Teach people', 'Carrie'),
        Task('Make some videos', 'Carrie'),
        Task('Inspire', 'Carrie'),

        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 ch3/b/tasks_proj
$ pytest
======================== test session starts ========================
collected 41 items

tests/func/test_add.py ...
tests/func/test_add_variety.py ........................
tests/func/test_api_exceptions.py .......
tests/func/test_unique_id.py .
tests/unit/test_Task.py .....

===================== 41 passed in 0.14 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 we wanted a test to use a fixture, we 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():

    def test_3(self):
        pass

    def test_4(self):
        pass
```

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 fixtures 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

```
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('%c', 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():
    time.sleep(1)

def test_2():
    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:

```
$ pytest -v -s test_autouse.py
======================== test session starts ========================
cachedir: .cache
collected 2 items

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

test_autouse.py::test_2 PASSED
test duration : 1.23 seconds
--
finished : Sun Apr 30 00:21:04 2017
-----------------

===================== 2 passed in 2.24 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

```
import pytest

@pytest.fixture(name='lue')
```

```
def fixture_with_a_name_much_longer_than_lue():
    'life, the universe, and everything'
    return 42

def test_everything_2(lue):
    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 file name 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 -- test_rename_fixture.py:4
    life, the universe, and everything

=================== 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 the filename and line number where they are defined. We can use this to look up the definition of lue.

Let's use that in the Tasks project:

```
$ cd ch3/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_just_a_few
    All summaries and owners are unique.
tasks_mult_per_owner
    Several owners with several tasks each.
tasks_db
    an empty tasks db
tasks_db_session
    initialize tasks db
db_with_3_tasks
    tasks db with 3 tasks, all unique
db_with_multi_per_owner
    tasks db 3 owners, all with 3 tasks

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

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

Parameterizing Fixtures

In *Parameterized Testing*, on page 40, we parameterized tests. We can also parameterize 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

```python
import pytest
import tasks
from tasks import Task

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

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

def equivelant(t1, t2):
    # someTask[:-1] -> a tuple containing (summary, owner, done)
    # i.e. compare everything but the id field
    return t1[:-1] == t2[:-1]
```

But now, instead of parameterizing the test, we'll parameterize a fixture called a_task:

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

```python
@pytest.fixture(params=tasks_to_try)
def a_task(request):
    'using no ids'
    return request.param

def test_add_a(tasks_db, a_task):
```

```
    task_id = tasks.add(a_task)
    t_from_db = tasks.get(task_id)
    assert equivelant(t_from_db, a_task)
```

The request listed in the fixture parameter is another builtin fixture that represents the calling state of the fixture. We'll explore it more in the next chapter. It has a field param that is filled in with one element from list assigned to params in @pytest.fixture(params=tasks_to_try).

This fixture is pretty simple—it just returns the reqeust.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:

```
$ pytest -v test_add_variety2.py::test_add_a
======================== test session starts ========================
cachedir: ../../.cache
collected 16 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 parameterized 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 list of ids'
    return request.param

def test_add_b(tasks_db, b_task):
    task_id = tasks.add(b_task)
    t_from_db = tasks.get(task_id)
    assert equivelant(t_from_db, b_task)
```

This gives us better identifiers:

```
$ pytest -v test_add_variety2.py::test_add_b
======================== test session starts ========================
cachedir: ../../.cache
collected 16 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(excercise,BrIaN,False)] PASSED
```

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

To do this we have to generate a list of ids ahead of time. With parameterized fixtures, there is another way—we can 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

```python
def id_func(fixture_value):
    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 id_func'
    return request.param

def test_add_c(tasks_db, c_task):
    task_id = tasks.add(c_task)
    t_from_db = tasks.get(task_id)
    assert equivelant(t_from_db, c_task)
```

The function will be called from the value of each item from the parameterization. Since the parameterization 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 on 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 ========================
cachedir: ../../.cache
collected 16 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(excercise,BrIaN,False)] PASSED

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

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

Parameterizing Fixtures in the Tasks Project

Now, let's see how we can use parameterized 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

@pytest.fixture(scope='session')
def tasks_db_session(tmpdir_factory):
    'initialize tasks db'
    # Setup : start db
    temp_dir = tmpdir_factory.mktemp('temp')
    tasks.start_tasks_db(str(temp_dir), 'tiny')

    yield # this is where the testing happens

    # Teardown : stop db
    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
    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

@pytest.fixture(scope='session', params=['tiny', 'mongo'])
```

```
def tasks_db_session(tmpdir_factory, request):
    # Setup : start db
    temp_dir = tmpdir_factory.mktemp('temp')
    tasks.start_tasks_db(str(temp_dir), request.param)

    yield # this is where the testing happens

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

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 parameterized tests or parameterized fixtures, pytest labels the different runs based on the value of the parameterization. And because the values are already strings, that works great.

Here's what we have so far:

```
$ pytest -v --tb=no
======================== test session starts ========================
cachedir: .cache
collected 27 items

tasks_proj/tests/func/test_add.py::test_add_returns_valid_id[tiny] PASSED
tasks_proj/tests/func/test_add.py::test_added_task_has_id_set[tiny] PASSED
tasks_proj/tests/func/test_add.py::test_add_increases_count[tiny] PASSED
tasks_proj/tests/func/test_api_exceptions.py::test_add_TypeError[tiny] PASSED
tasks_proj/tests/func/test_api_exceptions.py::test_list_TypeError[tiny] PASSED
tasks_proj/tests/func/test_api_exceptions.py::test_get_TypeError[tiny] PASSED
...
tasks_proj/tests/func/test_add.py::test_add_returns_valid_id[mongo] FAILED
tasks_proj/tests/func/test_add.py::test_added_task_has_id_set[mongo] FAILED
tasks_proj/tests/func/test_add.py::test_add_increases_count[mongo] PASSED
...
tasks_proj/tests/func/test_unique_id.py::test_unique_id[mongo] FAILED
tasks_proj/tests/unit/test_Task.py::test_asdict PASSED
tasks_proj/tests/unit/test_Task.py::test_from_dict PASSED
tasks_proj/tests/unit/test_Task.py::test_replace PASSED
tasks_proj/tests/unit/test_Task.py::test_defaults PASSED
tasks_proj/tests/unit/test_Task.py::test_member_access PASSED

================ 3 failed, 24 passed in 2.50 seconds ================
```

Hmm. Three failures. Looks like I'll need to do some debugging on the Mongo side. They are all around id. And Mongo has object id's instead of integers. D'oh! Looks like I'll have to read up on Mongo to figure out how to fix this, before I let anyone use the Mongo 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 of 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 we looked at pytest fixtures you write yourself, as well as a couple of builtin fixtures, tmpdir and tmpdir_factory. We'll take a closer look at the builtin fixtures in the next chapter.

CHAPTER 4

Builtin Fixtures

In the previous chapter, we looked at what fixtures are, how to write them, and how to use them for test data as well as setup and teardown code. We also utilized conftest.py for sharing fixtures between tests in multiple test files. By the end of Chapter 3, *pytest Fixtures*, on page 47, 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. We've already seen tmpdir and tmpdir_factory in use by the Tasks project in *Changing Scope for Tasks Project Fixtures*, on page 56. We'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 are 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 file name
    # 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.[1] 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*, on page 54, 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.

1. http://py.readthedocs.io/en/latest/path.html

To see how similar tmpdir and tmpdir_factory are 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()'
    # exmple 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:

```
$ pytest -q -s test_tmpdir.py::test_tmpdir_factory
base: /private/var/folders/53/zv4j_zc506x2xq25l31qxvxm0000gn\
        /T/pytest-of-okken/pytest-60
.
1 passed in 0.02 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

```
import pytest
import json

@pytest.fixture(scope='module')
def author_file_json(tmpdir_factory):
    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

```
import pytest
import json

def test_Brian_in_Portland(author_file_json):
    with author_file_json.open() as f:
        authors = json.load(f)
    assert authors['Brian']['City'] == 'Portland'

def test_all_have_cities(author_file_json):
    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, we'll look at how to add a custom command-line option and read the option value from within a test. We can read the value of command-line options directly from pytestconfig, but to add the option and have pytest parse it, we need to add a hook function. *Hook functions*, which are covered in more detail in Chapter 5, *Plugins*, on page 91, 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:

```
$ 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'))
```

We'll use pytestconfig again when demonstrating ini files in Chapter 6, *Configuration*, on page 107.

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. We'll 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:

```
$ pytest --verbose --tb=no test_pass_fail.py
==================== test session starts ====================
cachedir: .cache
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 ====================
cachedir: .cache
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 ====================
cachedir: .cache
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 parameterized test with one failure:

ch4/cache/test_few_failures.py

```python
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):
    sum_ = x + y
    assert sum_ == approx(expected)
```

And the output:

```
$ 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 are multiple failed cases you'd like to rerun, --lf is much easier. And if you are 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 =====================
cachedir: /path/to/ch4/cache/.cache
------------------------ cache values ------------------------
cache/lastfailed contains:
  {'test_pass_fail.py::test_this_fails': True}

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

Or you can look in the cache dir:

```
$ cat .cache/v/cache/lastfailed
{
  "test_pass_fail.py::test_this_fails": true
}(venv) $
```

For --lf and --ff, just don't pass in those flags if you want to ignore the cache for a test session. You can also pass in --clear-cache to clear the cache before the session.

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
```

```
    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 parameterized 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 we probably don't want to write a bunch of tests for this, I used random and parameterization 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:

```
$ pytest -q --cache-clear test_slower.py
.....
5 passed in 3.49 seconds
$ pytest -q --tb=line test_slower.py
...E..
========================== ERRORS ===========================
___________ ERROR at teardown of test_slow_stuff[2] ___________
E   AssertionError: test duration over 2x last duration
    assert 0.832009 <= (0.3676 * 2)
5 passed, 1 error in 2.51 seconds
```

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

```
$ pytest -q --cache-show
cachedir: /path/to/ch4/cache/.cache
------------------------ cache values ------------------------
cache/lastfailed contains:
  {'test_slower.py::test_slow_stuff[2]': True}
duration/test_slower.py::test_slow_stuff[0] contains:
  0.434069
duration/test_slower.py::test_slow_stuff[1] contains:
  0.354402
duration/test_slower.py::test_slow_stuff[2] contains:
  0.832009
```

```
duration/test_slower.py::test_slow_stuff[3] contains:
  0.425361
duration/test_slower.py::test_slow_stuff[4] contains:
  0.413355

no tests ran in 0.00 seconds
```

We 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[2] 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

2. https://github.com/pytest-dev/pytest/blob/master/_pytest/cacheprovider.py

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-show
cachedir: /path/to/ch4/cache/.cache
------------------------ cache values -------------------------
cache/lastfailed contains:
  {'test_slower_2.py::test_slow_stuff[1]': True}
duration/testdurations contains:
  {'test_slower_2.py::test_slow_stuff[0]': 0.468355,
   'test_slower_2.py::test_slow_stuff[1]': 0.296763,
   'test_slower_2.py::test_slow_stuff[2]': 0.977212,
   'test_slower_2.py::test_slow_stuff[3]': 0.05315,
   'test_slower_2.py::test_slow_stuff[4]': 0.277971}

no tests ran in 0.00 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 we have a function to print a greeting to stdout:

ch4/cap/test_capsys.py

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

We can't test it by checking the return value. We have to test stdout somehow. We 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 take a 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:

```
$ 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

```python
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 affect 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…”.[3] 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: re.sub('~', str(fake_home_dir), x)))
    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 a bit worried about what happens if the file already exists. We want to make 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: re.sub('~', str(fake_home_dir), x)))
    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
```

3. https://docs.python.org/3.6/library/os.path.html#os.path.expanduser

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 makes 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. We look at that in Chapter 7, *Using pytest with Other Tools*, on page 117.

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 don'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 ch4/dt/1
$ pytest  -v --doctest-modules --tb=short unnecessary_math.py
======================== test session starts ========================
cachedir: .cache
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:

```
$ pytest  -v --doctest-modules --tb=short unnecessary_math.py
======================== test session starts ========================
cachedir: .cache
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

```python
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.

We'll cover running doctest from pytest more in Chapter 7, *Using pytest with Other Tools*, on page 117.

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

```python
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

```python
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 are 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 ch3.
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 sec plus 2x last duration.
4. Run pytest with the modified fixture. Do the results seem reasonable?

What's Next

In this chapter, we looked at many of pytest's builtin fixtures. Next, we 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.

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'll 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*, on page 123 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 it's 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*, on page 123 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.

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 are 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 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*, on page 117.)

Note that with the local directory install method, you can install multiple versions and specify which one 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 are 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. 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.[1]

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 are 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*, on page 29, 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
```

1. http://doc.pytest.org/en/latest/_modules/_pytest/hookspec.html

```
@pytest.mark.usefixtures('tasks_db')
class TestAdd():

    def test_missing_summary(self):
        with pytest.raises(ValueError):
            tasks.add(Task(owner='bob'))

    def test_done_not_bool(self):
        with pytest.raises(ValueError):
            tasks.add(Task(summary='summary', done='True'))
```

Let's run them to see if they pass:

```
$ cd ch5/tasks_proj
$ pytest
========================= test session starts ==========================
collected 21 items

tests/func/test_add.py ...
tests/func/test_api_exceptions.py .F.......
tests/func/test_list.py ...
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 0x1039ffb70>

    def test_done_not_bool(self):
        with pytest.raises(ValueError):
>           tasks.add(Task(summary='summary', done='True'))
E           Failed: DID NOT RAISE <class 'ValueError'>

tests/func/test_api_exceptions.py:15: Failed
================= 1 failed, 20 passed in 0.13 seconds ==================
```

Let's run it again with -v for verbose. Since we have already seen the traceback, we can turn that off with --tb=no. And let's focus on the new tests with -k TestAdd, which works because there aren't any other tests with names that contain "TestAdd."

```
$ pytest -v --tb=no  -k TestAdd

================= 1 failed, 20 passed in 0.12 seconds ==================
$ pytest -v --tb=no -k TestAdd
========================= test session starts ==========================
cachedir: .cache
collected 21 items

tests/func/test_api_exceptions.py::TestAdd::test_missing_summary PASSED
tests/func/test_api_exceptions.py::TestAdd::test_done_not_bool FAILED

========================= 19 tests deselected ==========================
========== 1 failed, 1 passed, 19 deselected in 0.08 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 we can do with a pytest hook called pytest_report_header().

ch5/b/tasks_proj/tests/conftest.py
```
def pytest_report_header():
    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 user name 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/c/tasks_proj/tests/conftest.py
```
def pytest_report_teststatus(report):
    if 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 ch5/c/tasks_proj/tests/func
$ pytest --tb=no -k TestAdd
========================= test session starts ==========================
Thanks for running the tests.
collected 16 items

test_api_exceptions.py .O

========================= 14 tests deselected ==========================
========== 1 failed, 1 passed, 14 deselected in 0.07 seconds ===========
```

And the -v or --verbose flag will be nicer also:

```
$ pytest -v --tb=no -k TestAdd
========================= test session starts ==========================
cachedir: ../../.cache
Thanks for running the tests.
collected 16 items

test_api_exceptions.py::TestAdd::test_missing_summary PASSED
test_api_exceptions.py::TestAdd::test_done_not_bool OPPORTUNITY for improvement

========================= 14 tests deselected ==========================
========== 1 failed, 1 passed, 14 deselected in 0.08 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/d/tasks_proj/tests/conftest.py

```python
def pytest_addoption(parser):
    group = parser.getgroup('nice')
    group.addoption("--nice", action="store_true",
                    help="nice: turn failures into opportunities")

def pytest_report_teststatus(report):
    if pytest.config.option.nice and report.failed:
        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.[2]

We can manually test our plugin just by running it against our example file. First, with no --nice option, to make sure just the user name shows up:

```
$ pytest --tb=no -k TestAdd
========================= test session starts ==========================
Thanks for running the tests.
collected 21 items

tests/func/test_api_exceptions.py .F

========================= 19 tests deselected ==========================
========== 1 failed, 1 passed, 19 deselected in 0.09 seconds ===========
```

Now with --nice:

```
venv) $ pytest --tb=no -k TestAdd --nice
========================= test session starts ==========================
Thanks for running the tests.
collected 21 items

tests/func/test_api_exceptions.py .O

========================= 19 tests deselected ==========================
========== 1 failed, 1 passed, 19 deselected in 0.08 seconds ===========
```

And with --nice and --verbose:

```
$ pytest -v --tb=no -k TestAdd --nice
========================= test session starts ==========================
cachedir: .cache
Thanks for running the tests.
collected 21 items

tests/func/test_api_exceptions.py::TestAdd::test_missing_summary PASSED
tests/func/test_api_exceptions.py::TestAdd::test_done_not_bool \
```

2. https://docs.pytest.org/en/latest/writing_plugins.html

```
                                                OPPORTUNITY for improvement
========================= 19 tests deselected ==========================
========== 1 failed, 1 passed, 19 deselected in 0.08 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 and pretty easy to understand. 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.[3,4] 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 doesn't matter what you call it, but since we are making a plugin for the "nice" flag, let's call it "plugin-nice." We'll have two files in this new directory: pytest_nice.py, and setup.py. (The tests directory will be discussed in *Testing Plugins*, on page 100.)

```
pytest-nice
├── README.txt
├── 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

```python
import pytest

def pytest_report_header():
    if pytest.config.option.nice or pytest.config.getini('nice'):
        return "Thanks for running the tests."

def pytest_addoption(parser):
    group = parser.getgroup('nice')
    group.addoption("--nice", action="store_true",
```

3. http://python-packaging.readthedocs.io
4. https://www.pypa.io


```
                                help="nice: turn FAILED into OPPORTUNITY for improvement")

def pytest_report_teststatus(report):
    if pytest.config.option.nice and report.failed:
        return (report.outcome, 'O', 'OPPORTUNITY for improvement')
```

In setup.py, we need a very minimal call to setup():

ch5/pytest-nice/setup.py

```
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='MIT',
    py_modules=['pytest_nice'],
    entry_points={'pytest11': ['nice = pytest_nice', ], },
)
```

You'll want more information in your setup if you are 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',], },
```

We haven't talked about the README.txt 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.txt

```
# 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*, on page 94, 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

```
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(). We can pass in options if we want. The return value can then be examined further, and is of type RunResult.[5]

Usually, I look at stdout and ret. For checking the output like we did manually, use fnmatch_lines, passing in a list of regular expression strings that we want

5. https://docs.pytest.org/en/latest/writing_plugins.html#_pytest.pytester.RunResult

to see in the output, and then making sure that ret is 0 for passing sessions and 1 for failing sessions.

We can use our example file for more tests. So 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()
    result.stdout.fnmatch_lines([
        'Thanks for running the tests.'
    ])
```

```
    assert result.ret == 1
```

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 pytest-nice
$ ls
README.txt  pytest_nice.py  setup.py  tests/
$ pip install -e .
Obtaining file:///path/to/code/ch5/pytest-nice
Installing collected packages: pytest-nice
  Found existing installation: pytest-nice 0.1.0
    Uninstalling pytest-nice-0.1.0:
      Successfully uninstalled pytest-nice-0.1.0
  Running setup.py develop for pytest-nice
Successfully installed pytest-nice
$ pytest -v
========================== test session starts ==========================
cachedir: .cache
Thanks for running the tests.
plugins: nice-0.1.0
collected 6 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_help_message PASSED

======================= 6 passed in 0.37 seconds ========================
$ 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
```

Yay! All the tests pass. 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 pytest-nice/

$ python setup.py sdist
running sdist
...
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
  Running setup.py install for pytest-nice ... done
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 we 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 we've done some bug fixes and there are newer versions in myplugins, we 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.[6]

When you are contributing a pytest plugin, another great place to start is by using the cookiecutter-pytest-plugin[7]:

```
$ 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(). We used it in the Chapter 4 exercises as well. Now, let's make a plugin out of it.

1. Create a directory for "slower" similar to the directory described in *Creating an Installable Plugin*, on page 98.
2. Fill out all the files of the directory to make "slower" an installable plugin.
3. Write test code for the plugin. What kinds of test cases are needed?
4. Take a look at the Python Package Index[8] 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 plugin did you pick?

6. https://packaging.python.org/distributing
7. https://github.com/pytest-dev/cookiecutter-pytest-plugin
8. https://pypi.python.org/pypi

What's Next

We'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, we'll run through the different configuration files and learn what you can do there to make your testing life easier.

CHAPTER 6

Configuration

So far in this book, we've talked about the various non-test files that affect pytest mostly in passing, with the exception of conftest.py, which we covered quite thoroughly in Chapter 5, *Plugins*, on page 91. In this chapter, we'll take a look at the configuration files that affect pytest, discuss how they interact with pytest and what you can do with them, and make some changes to the configuration files of the Tasks project.

Understanding pytest Configuration Files

Before we discuss how you can alter pytest's default behavior, let's run down all 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*, on page 91.
- *__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*, on page 114.

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*, on page 117.

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*, on page 125.

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

```
... tox 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
...
```

We'll look at all of these settings in this chapter, except doctest_optionflags, which is covered in Chapter 7, *Using pytest with Other Tools*, on page 117.

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

We'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 bunch of the stack trace. It leaves the file and line number, though. The --strict option turns pytest warnings into errors and disallows markers to be used if they aren't registered in a config file. We'll see how to do that in the next section.

Registering Markers to Avoid Marker Typos

Custom markers, as discussed in *Marking Test Functions*, on page 30, 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, we can now also see them with pytest --markers with their descriptions:

```
$ 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 we use --strict, any misspelled markers show up as an error:

```
$ cd ch6/markers/tasks_proj
$ pytest --strict -m smoke
=================== test session starts =====================
collected 7 items / 2 errors

========================== ERRORS ==========================
__________ ERROR collecting tests/func/test_add.py ___________
tests/func/test_add.py:16: in <module>
    @pytest.mark.smoke
../../../../venv/lib/python3.6/site-packages/_pytest/mark.py:186: in __getattr__
    self._check(name)
../../../../venv/lib/python3.6/site-packages/_pytest/mark.py:201: in _check
    raise AttributeError("%r not a registered marker" % (name,))
E   AttributeError: 'smoke' not a registered marker
_____ ERROR collecting tests/func/test_api_exceptions.py _____
tests/func/test_api_exceptions.py:11: in <module>
    @pytest.mark.somke
../../../../venv/lib/python3.6/site-packages/_pytest/mark.py:186: in __getattr__
```

```
    self._check(name)
../../../../venv/lib/python3.6/site-packages/_pytest/mark.py:201: in _check
    raise AttributeError("%r not a registered marker" % (name,))
E   AttributeError: 'somke' not a registered marker
!!!!!!!!!! Interrupted: 2 errors during collection !!!!!!!!!!!
==================== 7 tests deselected =====================
=========== 7 deselected, 2 error in 0.21 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.

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, we 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. testpaths 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 we 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, we know that the root directory is going to be fixed, and we can list directories relative to that fixed root. These are also the cases where we really want to squeeze our 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 we've seen, testpaths 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 just 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*, on page 36.

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

```python
def test_a():
  pass
```

ch6/dups/b/test_foo.py

```python
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 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 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 not worry about it again.

Exercises

In Chapter 5, *Plugins*, on page 91, we 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. What tests should be added to test this ini file option?

5. Add the tests to the plugin tests directory. You'll need to look up some extra pytester functionality.[1]

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, we'll look at using pytest in conjunction with other powerful testing tools.

1. https://docs.pytest.org/en/latest/_modules/_pytest/pytester.html#Testdir

CHAPTER 7

Using pytest with Other Tools

APPENDIX 1

Virtual Environments

Python virtual environments allows you to set up a separate Python sandbox with its own set of packages separate from the system site-packages to work in. 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. You might find it handy to keep the dependent package requirements separate for every Python project you work on, and virtual environments let you do that.

Before using pip to install things, creating a virtual environment for a project is easy:

```
$ cd my_project
$ python3.6 -m venv your_virtual_env_name
```

That's it. Just run python3.6 -m venv your_virtual_env_name, and it creates a directory called your_virtual_env_name. The directory contains the subdirectories bin, include, and lib, with enough contents for this environment to run independently of the system environment and other virtual environments. The actual contents are system-dependent and dependent on what options you pass in. Go to python3.6 -m venv --help for more information on options.

To activate the virtual environment, use source your_virtual_env_name/bin/activate (your_virtual_env_name/bin/activate.bat in Windows), and deactivate to get out of it, like this:

```
$ cd my_project
$ source your_virtual_env_name/bin/activate
(your_virtual_env_name) $ pip list
pip (9.0.1)
setuptools (28.8.0)
(your_virtual_env_name) $ deactivate
$
```

(your_virtual_env_name) is added to your prompt to let you know that you are using a virtual environment. You can also put your virtual environment directory anywhere you want, but make sure you remember where it is. Personally, I don't want to think about naming too much, so I am putting the virtual environment directory right into my project directory, naming it venv, and using the --prompt option to change the prompt to something other than venv:

```
$ cd my_project
$ python3.6 -m venv --prompt my_project venv
$ source venv/bin/activate
(my_project) $
```

After setting up a virtual environment, you only have to remember three things:

- Where it is
- Use source path/to/my_project/venv/bin/activate to activate it
- Use deactivate to deactivate it

As simple as this may seem, these details are often hard to remember. Usually you remember the name of your project, so to make life easier, add something like this to your .bash_profile:

```
alias my_project="cd /path/to/my_project;source venv/bin/activate"
alias exit="deactivate"
```

Now, to work on a project, open a terminal window and type my_project. In this example, I changed directories to the project I want to work on and activated the virtual environment. exit normally closes my terminal window, which is never what I want. I kept typing exit instead of deactivate, so the exit alias is there to make life easier.

I have a bunch of different projects set up like this in my .bash_profile. A convenient side effect is that I can use grep activate ~/.bash_profile if I ever forget the names of the projects I'm working on and where their virtual environments are.

When venv Is Not Available

In Python versions prior to 3.3, venv is not available. In those versions, you can install virtualenv with pip and use in much the same way, just use -m virtualenv instead of -m venv.

APPENDIX 2

pip

Before playing with pytest, you need to make sure it's installed properly and that you are using an up-to-date version. The short answer for how to install pytest is to do a pip install, like this:

```
(venv) $ pip install -U pytest
```

There's a lot of hidden meaning in that short command line, however.

- `(venv)` shows that you're using a virtual environment called `venv`. This is optional.
- `$` shows that you're using bash or a bash-like terminal window to run your scripts.
- `pip` is Python's package manager.
- `install` tells `pip` that you want to install some package.
- `-U` tells `pip` that you want to make sure the package is upgraded, even if it's already installed. This is optional.
- `pytest` is the package you want `pip` to install.

It's worth taking a deeper look into each of these pieces, especially if they are new to you.

`pip` supposedly is a recursive acronym that stands for *Pip Installs Python* or *Pip Installs Packages.* (Programmers can be pretty nerdy with their humor.) `pip` is installed as part of your Python installation. 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 Python Package Index (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 it 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 more cool stuff like setting up scripts defined by the package, wheel caching, and more.

As an example of how pip works, I have both Python 3.5 and 3.6 installed and they live in /usr/local/bin as python3.5 and python3.6.

I know this because I ran the following:

```
$ which python
/usr/local/bin/python
$ ls /usr/local/bin/pyth*
...
/usr/local/bin/python3.5@
...
/usr/local/bin/python3.6@
...
```

I can install into either of them with a slight change in the command:

```
$ python3.5 -m pip install something
$ python3.6 -m pip install something
```

This installs something in both versions of Python. Actually, something shows up in the same site-packages directory as pip:

```
$ python3.5 -m pip --version
pip 8.1.2 from
 /Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages
$ python3.6 -m pip --version
pip 9.0.1 from
 /Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages
```

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. 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, but it always sticks the packages in site-packages unless you are using Python virtual environments.

APPENDIX 3

Plugin Sampler Pack

APPENDIX 4

Packaging and Distributing Python Projects

APPENDIX 5

xUnit Fixtures

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page
https://pragprog.com/book/bopytest
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
https://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
https://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
https://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: *https://pragprog.com/book/bopytest*

Contact Us

Online Orders:	*https://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://write-for-us.pragprog.com*
Or Call:	+1 800-699-7764