Extracted from:

Python Testing with pytest

Simple, Rapid, Effective, and Scalable

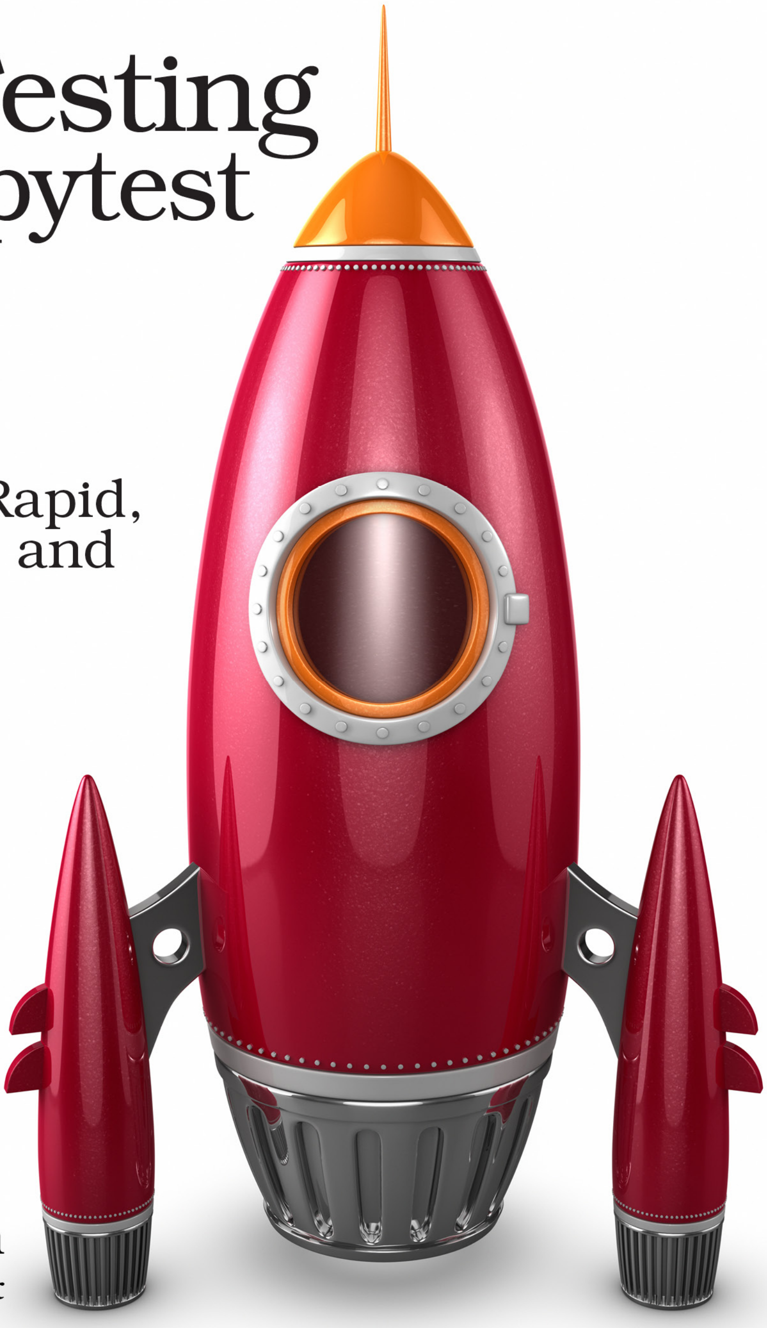
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Python Testing with pytest

Simple, Rapid, Effective, and Scalable



Brian Okken

edited by Katharine Dvorak

Python Testing with pytest

Simple, Rapid, Effective, and Scalable

Brian Okken

The Pragmatic Bookshelf

Raleigh, North Carolina

The team that produced this book includes:

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

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

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

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

Running pytest

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

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

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

ch1/tasks/test_three.py

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

from collections import namedtuple

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


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


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

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

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

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

ch1/tasks/test_four.py

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

from collections import namedtuple


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

```
Task.__new__.__defaults__ = (None, None, False, None)

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

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

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

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

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

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

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

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

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

```
$ pytest tasks/test_three.py tasks/test_four.py
```

```
===================== test session starts ======================
collected 4 items

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

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

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

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

test_four.py ..
test_three.py ..

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

The part of pytest execution where pytest goes off and finds which tests to run is called *test discovery*. pytest was able to find all the tests we wanted it to run because we named them according to the pytest naming conventions. Here's a brief overview of the naming conventions to keep your test code discoverable by pytest:

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

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

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

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

test_three.py ..

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

The output tells us quite a bit.

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

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

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

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

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

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

collected 2 items

These are the two test functions in the file.

test_three.py ..

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

== 2 passed in 0.01 seconds ==

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

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

Here are the possible outcomes of a test function:

- PASSED (.): The test ran successfully.
- FAILED (F): The test did not run successfully (or XPASS + strict).

- SKIPPED (s): The test was skipped. You can tell pytest to skip a test by using either the @pytest.mark.skip() or pytest.mark.skipif() decorators, discussed in *Skipping Tests*, on page ?.
- 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 ?.
- 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 ?, or in a hook function, discussed in Chapter 5, *Plugins*, on page ?.