Extracted from:

Python Testing with pytest

Simple, Rapid, Effective, and Scalable

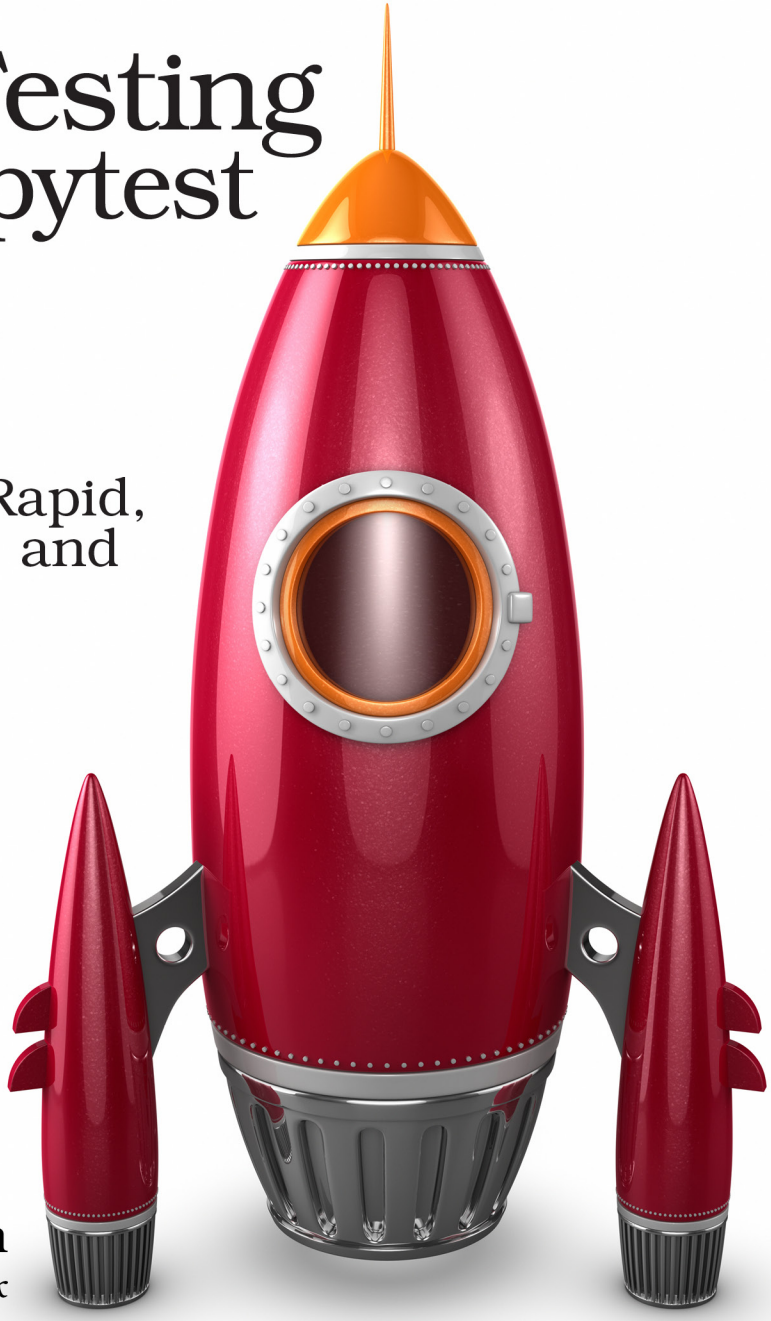
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Python Testing with pytest

Simple, Rapid, Effective, and Scalable



Brian Okken

edited by Katharine Dvorak

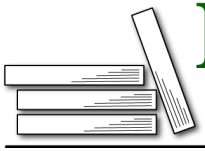
Python Testing with pytest

Simple, Rapid, Effective, and Scalable

Brian Okken

The Pragmatic Bookshelf

Raleigh, North Carolina



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

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

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 ?, we tested it manually by using a sample test file, running pytest against it, and looking at the output to make sure it was right. We can do the same thing in an automated way using a plugin called pytester that ships with pytest but is disabled by default.

Our test directory for pytest-nice has two files: conftest.py and test_nice.py. To use pytester, we need to add just one line to conftest.py:

ch5/pytest-nice/tests/conftest.py

```
"""pytester is needed for testing plugins."""
pytest_plugins = 'pytester'
```

This turns on the pytester plugin. We will be using a fixture called testdir that becomes available when pytester is enabled.

Often, tests for plugins take on the form we've described in manual steps:

1. Make an example test file.
2. Run pytest with or without some options in the directory that contains our example file.
3. Examine the output.
4. Possibly check the result code—0 for all passing, 1 for some failing.

Let's look at one example:

ch5/pytest-nice/tests/test_nice.py

```
def test_pass_fail(testdir):

    # create a temporary pytest test module
    testdir.makepyfile("""
        def test_pass():
            assert 1 == 1

        def test_fail():
            assert 1 == 2
    """)

    # run pytest
    result = testdir.runpytest()

    # fnmatch_lines does an assertion internally
    result.stdout.fnmatch_lines([
        '*.F',  # . for Pass, F for Fail
    ])

    # make sure that that we get a '1' exit code for the testsuite
```

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

The testdir fixture automatically creates a temporary directory for us to put test files. It has a method called makepyfile() that allows us to put in the contents of a test file. In this case, we are creating two tests: one that passes and one that fails.

We run pytest against the new test file with testdir.runpytest(). You can pass in options if you want. The return value can then be examined further, and is of type RunResult.[5]

Usually, I look at stdout and ret. For checking the output like we did manually, use fnmatch_lines, passing in a list of strings that we want to see in the output, and then making sure that ret is 0 for passing sessions and 1 for failing sessions. The strings passed into fnmatch_lines can include glob wildcards. We can use our example file for more tests. Instead of duplicating that code, let's make a fixture:

ch5/pytest-nice/tests/test_nice.py

```
@pytest.fixture()
def sample_test(testdir):
    testdir.makepyfile("""
        def test_pass():
            assert 1 == 1

        def test_fail():
            assert 1 == 2
    """)
    return testdir
```

Now, for the rest of the tests, we can use sample_test as a directory that already contains our sample test file. Here are the tests for the other option variants:

ch5/pytest-nice/tests/test_nice.py

```
def test_with_nice(sample_test):
    result = sample_test.runpytest('--nice')
    result.stdout.fnmatch_lines(['*.O', ])  # . for Pass, O for Fail
    assert result.ret == 1

def test_with_nice_verbose(sample_test):
    result = sample_test.runpytest('-v', '--nice')
    result.stdout.fnmatch_lines([
        '*::test_fail OPPORTUNITY for improvement',
    ])
    assert result.ret == 1

def test_not_nice_verbose(sample_test):
    result = sample_test.runpytest('-v')
    result.stdout.fnmatch_lines(['*::test_fail FAILED'])
    assert result.ret == 1
```

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

Just a couple more tests to write. Let's make sure our thank-you message is in the header:

ch5/pytest-nice/tests/test_nice.py
```
def test_header(sample_test):
    result = sample_test.runpytest('--nice')
    result.stdout.fnmatch_lines(['Thanks for running the tests.'])

def test_header_not_nice(sample_test):
    result = sample_test.runpytest()
    thanks_message = 'Thanks for running the tests.'
    assert thanks_message not in result.stdout.str()
```

This could have been part of the other tests also, but I like to have it in a separate test so that one test checks one thing.

Finally, let's check the help text:

ch5/pytest-nice/tests/test_nice.py
```
def test_help_message(testdir):
    result = testdir.runpytest('--help')

    # fnmatch_lines does an assertion internally
    result.stdout.fnmatch_lines([
        'nice:',
        '*--nice*nice: turn FAILED into OPPORTUNITY for improvement',
    ])
```

I think that's a pretty good check to make sure our plugin works.

To run the tests, let's start in our pytest-nice directory and make sure our plugin is installed. We do this either by installing the .zip.gz file or installing the current directory in editable mode:

```
$ cd /path/to/code/ch5/pytest-nice/
$ pip install .
Processing /path/to/code/ch5/pytest-nice
Requirement already satisfied: pytest in
    /path/to/venv/lib/python3.6/site-packages (from pytest-nice==0.1.0)
Requirement already satisfied: py>=1.4.33 in
    /path/to/venv/lib/python3.6/site-packages (from pytest->pytest-nice==0.1.0)
Requirement already satisfied: setuptools in
    /path/to/venv/lib/python3.6/site-packages (from pytest->pytest-nice==0.1.0)
Building wheels for collected packages: pytest-nice
  Running setup.py bdist_wheel for pytest-nice ... done
  ...
Successfully built pytest-nice
Installing collected packages: pytest-nice
Successfully installed pytest-nice-0.1.0
```

Now that it's installed, let's run the tests:

```
$ pytest -v
====================== test session starts ======================
plugins: nice-0.1.0
collected 7 items

tests/test_nice.py::test_pass_fail PASSED
tests/test_nice.py::test_with_nice PASSED
tests/test_nice.py::test_with_nice_verbose PASSED
tests/test_nice.py::test_not_nice_verbose PASSED
tests/test_nice.py::test_header PASSED
tests/test_nice.py::test_header_not_nice PASSED
tests/test_nice.py::test_help_message PASSED

=================== 7 passed in 0.34 seconds ====================
```

Yay! All the tests pass. We can uninstall it just like any other Python package or pytest plugin:

```
$ pip uninstall pytest-nice
Uninstalling pytest-nice-0.1.0:
  /path/to/venv/lib/python3.6/site-packages/pytest-nice.egg-link
  ...
Proceed (y/n)? y
  Successfully uninstalled pytest-nice-0.1.0
```

A great way to learn more about plugin testing is to look at the tests contained in other pytest plugins available through PyPI.

Creating a Distribution

Believe it or not, we are almost done with our plugin. From the command line, we can use this setup.py file to create a distribution:

```
$ cd /path/to/code/ch5/pytest-nice
$ python setup.py sdist
running sdist
running egg_info
creating pytest_nice.egg-info
...
running check
creating pytest-nice-0.1.0
...
creating dist
Creating tar archive
...
$ ls dist
pytest-nice-0.1.0.tar.gz
```

(Note that sdist stands for “source distribution.”)

Within pytest-nice, a dist directory contains a new file called pytest-nice-0.1.0.tar.gz. This file can now be used anywhere to install our plugin, even in place:

```
$ pip install dist/pytest-nice-0.1.0.tar.gz
Processing ./dist/pytest-nice-0.1.0.tar.gz
...
Installing collected packages: pytest-nice
Successfully installed pytest-nice-0.1.0
```

However, you can put your .tar.gz files anywhere you'll be able to get at them to use and share.

Distributing Plugins Through a Shared Directory

pip already supports installing packages from shared directories, so all we have to do to distribute our plugin through a shared directory is pick a location we can remember and put the .tar.gz files for our plugins there. Let's say we put pytest-nice-0.1.0.tar.gz into a directory called myplugins.

To install pytest-nice from myplugins:

```
$ pip install --no-index --find-links myplugins pytest-nice
```

The --no-index tells pip to not go out to PyPI to look for what you want to install. The --find-links myplugins tells PyPI to look in myplugins for packages to install. And of course, pytest-nice is what we want to install.

If you've done some bug fixes and there are newer versions in myplugins, you can upgrade by adding --upgrade:

```
$ pip install --upgrade --no-index --find-links myplugins pytest-nice
```

This is just like any other use of pip, but with the --no-index --find-links myplugins added.

Distributing Plugins Through PyPI

If you want to share your plugin with the world, there are a few more steps we need to do. Actually, there are quite a few more steps. However, because this book isn't focused on contributing to open source, I recommend checking out the thorough instruction found in the Python Packaging User Guide.[6]

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

```
$ pip install cookiecutter
```

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

```
$ cookiecutter https://github.com/pytest-dev/cookiecutter-pytest-plugin
```

This project first asks you some questions about your plugin. Then it creates a good directory for you to explore and fill in with your code. Walking through this is beyond the scope of this book; however, please keep this project in mind. It is supported by core pytest folks, and they will make sure this project stays up to date.

Exercises

In ch4/cache/test_slower.py, there is an autouse fixture called check_duration(). You used it in the Chapter 4 exercises as well. Now, let's make a plugin out of it.

1. Create a directory named pytest-slower that will hold the code for the new plugin, similar to the directory described in *Creating an Installable Plugin*, on page ?.
2. Fill out all the files of the directory to make pytest-slower an installable plugin.
3. Write some test code for the plugin.
4. Take a look at the Python Package Index[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's Next

You've used conftest.py a lot so far in this book. There are also configuration files that affect how pytest runs, such as pytest.ini. In the next chapter, you'll run through the different configuration files and learn what you can do there to make your testing life easier.

8. https://pypi.python.org/pypi