Extracted from:

Intuitive Python

Productive Development for Projects that Last

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic exPress

Intuitive Python

Productive Development for Projects that Last

David Muller

edited by Adaobi Obi Tulton

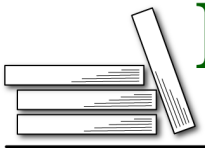
Intuitive Python

Productive Development for Projects that Last

David Muller

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Adaobi Obi Tulton
Copy Editor: Karen Galle
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

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

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

ISBN-13: 978-1-68050-823-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—June 2021

Keeping Your Source Organized

Just like the dinosaurs in *Jurassic Park*, Python codebases have a habit of breaking out from inside the fenced pens we try to create for them. One particularly vexing problem can be keeping project code organized. As time passes, code starts to live in places it shouldn't and it can become increasingly difficult to find what files code should live in or where new code should be placed. In this section, we'll explore a general strategy that uses Python's built-in unittest module to keep a directory organized.

Maintaining Organization in a tests/ Directory

Many Python codebases contain tests to help verify that they are working correctly. Frequently, these tests are defined in a directory named tests/ at the top level of the codebase. The tests/ directory structure typically matches the directory structure in the corresponding source directory that is being tested.

Let's consider a Python project with the following directory structure:

```
├── j_park
│   ├── __init__.py
│   ├── dinosaurs
│   │   ├── __init__.py
│   │   ├── raptor.py
│   │   └── t_rex.py
│   └── fences
│       ├── __init__.py
│       ├── cable.py
│       └── electrified.py
└── tests
    ├── __init__.py
    ├── fences
    │   ├── __init__.py
    │   └── test_cable.py
    ├── test_electrified.py
    └── test_raptor.py
```

This source tree has a j_park/ directory with source code and a tests/ directory with test_*.py files that correspond to source files in j_park/.

If you look at this tree, you'll see that the layout of the j_park/ directory has diverged from the layout in tests/. Among other inconsistencies, test_electrified.py is at the root level of tests/ instead of in the tests/fences/ subdirectory. Additionally, test_raptor.py is orphaned at the root level of tests/ instead of occupying a place in a tests/dinosaurs/ subdirectory.

Python unittest Test Discovery Still Requires __init__.py

In Python 3, you generally do not need to create _init_.py files like you did in Python 2. The _init_.py files are included in this example, however, because a longstanding bug prevents Python unittest discovery (for example, python3 -m unittest discover --help) from finding test files that don't have _init_.py siblings.[10]

With the existing disorganization, it would be unsurprising to receive a change request that updates the directory layout so that jurassic_park/ and tests/ diverge further:

```
├── jurassic_park
│   ├── __init__.py
│   ├── dinosaurs
│   │   ├── __init__.py
│   │   ├── raptor.py
│   │   └── t_rex.py
│   └── fences
│       ├── __init__.py
│       ├── cable.py
│       └── electrified.py
└── tests
    ├── __init__.py
    ├── fences
    │   ├── __init__.py
    │   └── test_cable.py
    ├── test_electrified.py
    ├── test_raptor.py
+   └── test_t_rex.py
```

Notice in the example that a new file test_t_rex.py has been added directly under tests/. This doesn't match j_park/ where t_rex.py lives under the dinosaurs/ subdirectory.

The tests/ directory is becoming increasingly divergent from the j_park/ source directory. As the project accumulates more files, it becomes harder and harder to find where a source code is tested. It's true that this disorganization may not be the end of the world, but it increases the barrier to entry to your project. Where does test code live? Where should new contributors find tests? How do they know if something is already tested? You notice this and may feel a bit wary—let's address this wariness and capitalize on the opportunity to improve the code's organization.

10. https://stackoverflow.com/a/53976736

You can mandate that the j_park/ and tests/ directory layouts match using a unittest TestCase in a new file tests/test_directory_layout.py:

test_directory_layout.py

```
import unittest
from pathlib import Path

class TestDirectoryLayout(unittest.TestCase):
    def test_tests_layout_matches_j_park(self):
        # verify that this file is - itself - in tests/
        this_files_path = Path(__file__)
        tests_dir = this_files_path.parent
        self.assertEqual(tests_dir.name, "tests")

        # get a path to the j_park/ source directory
        j_park_path = Path(tests_dir.parent, "j_park")

        # loop through all test_*.py files in tests/
        # (and its subdirectories)
        for test_file_path in tests_dir.glob("**/test_*.py"):
            # skip this file: we don't expect there to be a
            # corresponding source file for this layout enforcer
            if test_file_path == this_files_path:
                continue

            # construct the expected source_path
            source_rel_dir = test_file_path.relative_to(tests_dir).parent
            source_name = test_file_path.name.split("test_", maxsplit=1)[1]
            source_path = Path(j_park_path, source_rel_dir, source_name)

            error_msg = (
                f"{test_file_path} found, but {source_path} missing."
            )
            self.assertTrue(source_path.is_file(), msg=error_msg)
```

The TestCase class TestDirectoryLayout defines a single test method named test_tests_layout_matches_j_park. Using the pathlib standard library module, the test method loops through every test_*.py in the tests/ directory and ensures that the test_*.py file corresponds to a source file. If some of the functions used in the test shown look unfamiliar, that's OK. The most important thing to keep in mind is the strategy of writing a test case that forces you and your teammates to follow a pattern and stay organized—that you keep holes out of your fences. I encourage you to adapt the test into any of your own projects.

If you were to duplicate the preceding directory structure and run the test using the unittest module

```
python3 -m unittest tests/test_directory_layout.py
```

you might see a failure message like the following:

```
AssertionError: False is not true : /home/user/code/tests/test_raptor.py
```

```
found, but /home/user/code/j_park/raptor.py missing.
```

The output indicates that test_raptor.py does not correspond to an actual source file: it is an orphan. You have successfully added a test that automatically detects when other tests are out of position and do not correspond to a source file.

If you repeatedly run the test—fixing the failure messages as you go—your tests/ directory will eventually match the layout of j_park/ and will be continually enforced. The eventual output of your corrections will look like the following:

```
├── j_park
│   ├── __init__.py
│   ├── dinosaurs
│   │   ├── __init__.py
│   │   ├── raptor.py
│   │   └── t_rex.py
│   └── fences
│       ├── __init__.py
│       ├── cable.py
│       └── electrified.py
└── tests
    ├── __init__.py
    ├── dinosaurs
    │   ├── __init__.py
    │   ├── test_raptor.py
    │   └── test_t_rex.py
    └── fences
        ├── __init__.py
        ├── test_cable.py
        └── test_electrified.py
```

Think of other ways you can add unit tests that improve your day-to-day living experience in a code base. It might be useful to, for example, write a unit test that enforces any configuration files, CSV files, and so on in your code base to store their contents in alphabetical order. Having their contents in order makes the files more pleasant to read and edit. Anytime you catch yourself writing a comment like # Please keep this list in alphabetical order, consider using Python's high level tooling to write a test that mandates the constraint instead.

In the final section of this chapter, we'll explore a risk a little more direct than disorganized file systems: wildcard variable shadowing.