5 Building and maintaining a test suite

This chapter covers

- Running unit tests with pytest
- Creating test coverage reports with pytest-cov
- Reducing duplicated test code with parameterization
- Automating packaging for testing using tox
- Creating a test matrix

Tests are an important aspect of any project you plan to maintain. They can ensure that new functionality behaves as you expect and that existing functionality hasn't regressed. Tests are the guardrails for refactoring code—a common activity as projects mature.

With all this value that tests provide, you might think all open source packages would be thoroughly tested. But many projects pass on things like code coverage or testing for multiple target platforms because of the maintenance burden they present. Some maintainers even *create* maintenance burden without realizing it due to the way they design and run their test suite. In this chapter, you'll learn some beneficial aspects of testing and how to introduce them to your package's test suite, with an eye toward automation and scalability.

If you're still new to unit-testing concepts, you can learn all about them in Roy Osherove's *The Art of Unit Testing*, 3rd ed. (Manning Publications, anticipated 2023, http://mng.bz/YKGj).

IMPORTANT You can use the code companion (http://mng.bz/69A5) to check your work for the exercises in this chapter.

5.1 Integrating a testing setup

The first step toward building a robust test suite is configuring a *test runner* to run any tests for the project. If you've used the built-in `unittest` module in the past, you've most likely used a command like `python -m unittest discover` as your test runner. `unittest` is a perfectly capable piece of software, but, like any Python built-in, it requires work on your part when you want to extend or change its behavior. Further, the framework `unittest` employs is inspired both functionally and semantically by the xUnit (https://xunit.net/) family of testing frameworks, which can feel awkward because its conventions don't always follow PEP 8 (https://www.python.org/dev/peps/pep-0008/) style.

For a testing experience that aligns more closely with Python runtime code and can scale in productivity with your test suite, pytest (https://docs.pytest.org) is a strong alternative. You'll use pytest throughout the rest of this chapter and learn some of the advantages it has over the `unittest` module.

5.1.1 The pytest testing framework

pytest aims to make it easier to write simple tests and support increasingly complex projects as they grow. It can run `unittest`-based test suites out of the box but also provides its own assertion syntax and a plugin-based architecture to extend and change its behavior to suit your needs. The framework also provides a number of utilities for designing scalable tests, such as

- *Test fixtures*—Functions that provide additional dependencies to a test, such as data or database connections
- *Parameterized tests*—The ability to write a single test function and multiple sets of input arguments to create a unique test for each set of inputs

TIP For an in-depth look at pytest and all its features, check out Brian Okken's *Python Testing with pytest*, 2nd ed. (Pragmatic Bookshelf, 2022, http://mng.bz/1olg).

You *must* install pytest in the same virtual environment where your package and its dependencies are installed. Unit tests execute your real code, and that code must be importable. As an example, if you install pytest globally using pipx, pytest won't know where to find your project's dependencies and will fail to import them. Jump straight into pytest by installing it into your project's virtual environment using the following command:

```
$ py -m pip install pytest
```

Installing pytest makes the `pytest` module available. In chapter 4, you installed your package's code into the virtual environment so that it could be imported. pytest imports your code the same way when running tests. Run pytest now using the following command:

```
$ py -m pytest
```

This causes pytest to discover any tests it can and then execute them. Because you don't have any tests yet, you will see output like the following:

Environment synopsis showing the Python version, pytest version, and plugin versions

Directory for configuration, test discovery, and so on

```
=============== test session starts ===============
platform darwin -- Python 3.10.0b2+,
➥ pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: /path/to/first-python-package
collected 0 items

============== no tests ran in 0.00s ==============
```

No tests were discovered.

No tests were executed.

Remember that in chapter 3 you created a layout for your project that separates the source code from the test code. You added your implementation code to the src/ directory and created an empty test/ directory. To avoid including tests in the packaged code and to keep your tests in one easy-to-find place, you should place your tests in the test/ directory. By default, pytest discovers tests anywhere they might exist in your project. This includes tests in the root directory of the project or in the src/ directory, which isn't ideal. You'll configure pytest to ensure that it runs only tests placed in the proper location.

Exercise 5.1

Create a `test_harmonic_mean.py` module in the root directory of the project, and add one test function called `test_always_passes` that always passes. If you aren't familiar with pytest, you can use Python `assert` statements directly for your test assertions; a statement like `assert True` will always pass.

After you create the test module, run pytest again. This time you will see output like the following:

```
=============== test session starts ===============
...
collected 1 item

test_harmonic_mean.py .

================ 1 passed in 0.04s ================
```

One test was discovered.

A list of test modules discovered and a dot for each passing test

One test was executed in 0.04 seconds, and it passed.

This demonstrates that pytest is looking everywhere under the project's root directory for tests. To encourage the placement of tests in the appropriate location, you should configure pytest to look only in the test/ directory. You can add configuration for pytest into your package's setup.cfg file using a new section called `[tool:pytest]`. The `testpaths` key maps to a list of paths in which to look for tests. You need just one: `test`. After you add this configuration, pytest should confirm in its output both that it's using setup.cfg as the configuration file and that it found the `testpaths` configuration.

Exercise 5.2

Add the pytest configuration in `setup.cfg` to look only in the test/ directory for tests. After you add the configuration, do the following:

- Run pytest again, and confirm that it discovers and runs no tests.
- Move the `test_harmonic_mean.py` module into the test/ directory where it belongs.
- Run pytest another time, and confirm that it discovers and runs the test you wrote.

Now you're in a good place to add more tests. pytest will pick up any new test modules you add to the test/ directory, according to its naming conventions, as follows (see figure 5.1):

1. Start in any directory in `testpaths`.
2. Find modules named `test_*.py`.
3. Find classes in those modules named `Test*`.
4. Find functions in those modules, or methods in those classes, named `test_*`.

Now that you've created a mechanism for writing and running tests, the next step is figuring out which tests to write. In the next section, you'll integrate test coverage and write more tests to ensure you're covering all the code paths for your package.

5.1.2 Adding test coverage measurement

Before starting into test coverage, you must first understand that it isn't a silver bullet. Test coverage tells you how much of your runtime code executed during test execution and can even measure how many conditional branches executed. But test coverage doesn't ensure that all those lines and branches have corresponding assertions that verify their behavior. A test that executes your entire code base but ends with an `assert True` will have 100% coverage but no value whatsoever.

That said, if you're diligent about designing your test cases properly, coverage is a useful tool to help you find areas of code that definitely *don't* have any assertions made about them. You can use this to add valuable tests and refactor your suite to get better

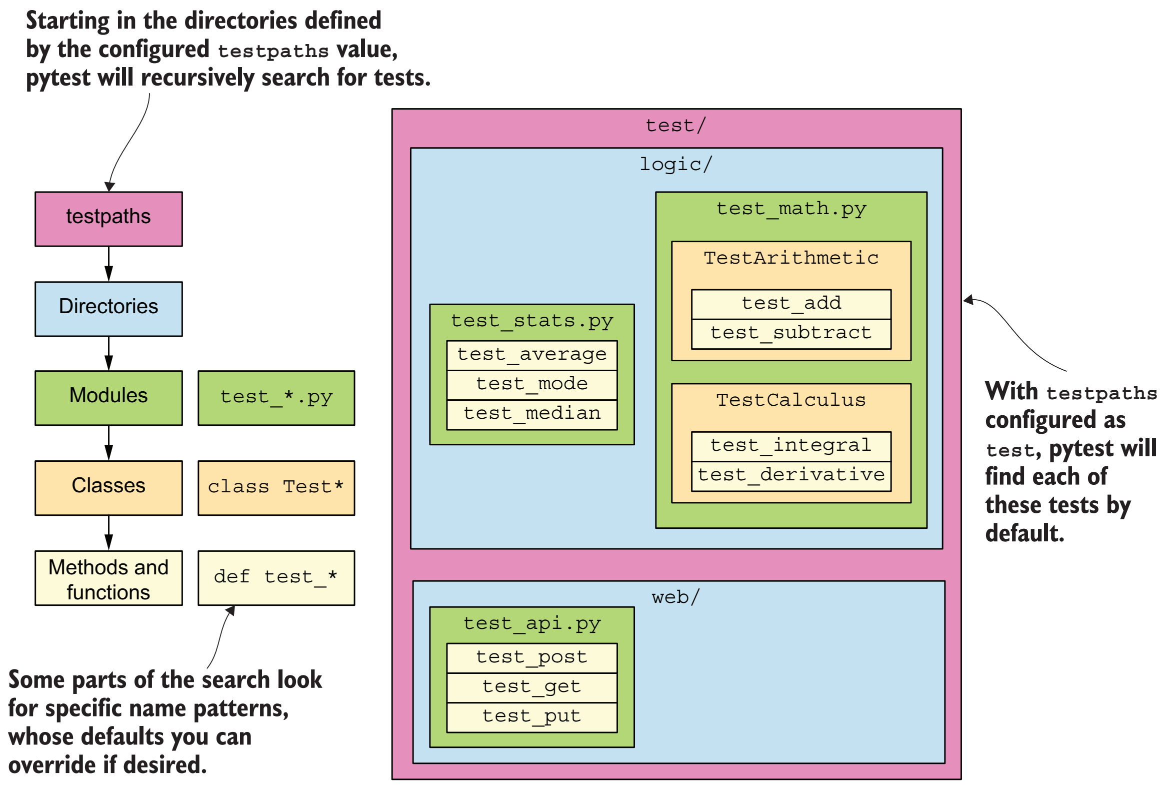
Starting in the directories defined by the configured testpaths value, pytest will recursively search for tests.
testpaths
Directories
Modules
test_*.py
Classes
class Test*
Methods and functions
def test_*
Some parts of the search look for specific name patterns, whose defaults you can override if desired.
test/
logic/
test_stats.py
test_average
test_mode
test_median
test_math.py
TestArithmetic
test_add
test_subtract
TestCalculus
test_integral
test_derivative
web/
test_api.py
test_post
test_get
test_put
With testpaths configured as test, pytest will find each of these tests by default.

Figure 5.1 pytest discovers the unit tests in a project using recursive pattern matching.

coverage of your runtime code. To start measuring coverage, install the `pytest-cov` package in your project's virtual environment like so:

```
$ py -m pip install pytest-cov
```

This package provides a pytest plugin that integrates the Coverage.py project (https://coverage.readthedocs.io/) so that you can run it using pytest. Coverage.py is the de facto standard for measuring Python code coverage. With `pytest-cov` installed, run pytest with the `--cov` option to collect coverage measurements. You will see additional output at the end of the execution after the usual pytest report you've been seeing, listing a lot of files that you've never heard of, as shown here:

```
$ py -m pytest --cov
=============== test session starts ===============
...

-- coverage: platform darwin, python 3.10.0-beta-2
Name                    Stmts   Miss Branch BrPart  Cover
-----------------------------------------------------
... A lot of files ...
-----------------------------------------------------
```

Like pytest, Coverage.py prints some environment info.

The name, line, branch, and overall coverage for each file

Coverage.py is measuring a lot of files that aren't yours.

```
TOTAL
================ 1 passed in 0.04s ================
```

The overall coverage of the entire code base

Coverage.py measures the coverage of all the installed Python code it can find, which includes code for your package's dependencies and even pytest itself. To only measure coverage of your package, you can specify your import package's name as the value for the `--cov` option. Your tests don't even import your package yet, so you should expect the coverage to be zero. Run pytest again with your package specified for coverage and confirm this is the case. Coverage.py will produce output like the following:

```
$ py -m pytest --cov=imppkg
=============== test session starts ===============
...

Coverage.py warning: Module imppkg was never imported.
➥ (module-not-imported)
Coverage.py warning: No data was collected.
➥ (no-data-collected)
WARNING: Failed to generate report: No data to report.

/path/to/first-python-package/.venv/lib/python3.10/
➥ site-packages/pytest_cov/plugin.py:285:
➥ PytestWarning: Failed to generate report:
➥ No data to report.

  warnings.warn(pytest.PytestWarning(message))


-- coverage: platform darwin, python 3.10.0-beta-2 --


================ 1 passed in 0.04s ================
```

A confirmation that your package isn't imported in the tests

A confirmation that your runtime code is not covered at all

You can quickly fix the `module-not-imported` issue by importing your code in your tests. At the top of the `test_harmonic_mean.py` module, import the `harmonic_mean` function and the `main` function that supports the `harmony` command. After you add the imports, run pytest with coverage again. This time, you will see the `__init__.py` and `harmony.py` modules in the coverage output, similar to the following:

```
$ py -m pytest --cov=imppkg
...

-- coverage: platform darwin, python 3.10.0-beta-2 --
Name                    Stmts   Miss   Cover
-----------------------------------------
.../__init__.py             0      0    100%
.../harmony.py              6      2     67%
----------------------------------------
TOTAL                       6      2     67%
```

There's no code in this module, so it's fully covered.

There are six statements in this module, two of which aren't executed by tests.

You should be able to see clearly now that test coverage doesn't necessarily correlate with test value. You've written one test that doesn't exercise any code, and you already have 67% coverage of your Python modules.

Coverage for non-Python extensions

Coverage.py typically covers Python source code, but for some non-Python extensions, it may be possible to cover their source code by enabling line tracing during compilation and using Coverage.py plugins that understand the line trace information. As an example, you can specify additional directives in your Cython .pyx files to enable line tracing and use the `Cython.Coverage` plugin to measure coverage.

Enabling branch coverage

In addition to line coverage, an important aspect of testing is understanding how many alternative execution paths are possible and which of those paths are untested. A piece of code's *cyclomatic complexity* (Thomas J. McCabe, "A Complexity Measure." *IEEE Transactions on Software Engineering* 4 [1976]: 308–20., doi:10.1109/tse.1976.233837) measures the number of paths through the code, and for full coverage of your code's behavior, you need a test for each path. In Coverage.py, this is known as *branch coverage*.

To configure branch coverage for your tests, add a new section to setup.cfg called `[coverage:run]`. In this section, add a `branch` key with a value of `True` (see listing 5.1). This produces two new columns in the coverage output:

- *Branch*—How many branches exist throughout the code
- *BrPart*—How many branches are only partially covered by tests

Exercise 5.3

While you're adding the `[coverage:run]` section, add a `source` key with a value of `imppkg`. This is a handy way to stop specifying `imppkg` to the `--cov` option for pytest each time and ensures that anyone running tests with coverage will see the same output. You can also avoid specifying `--cov` altogether by adding an `addopts` key to the `[tool:pytest]` section with a value of `--cov`. You can override this at the command line later as desired using the corresponding `--no-cov` option.

After adding those configurations, what command should you run to get the same behavior you have been so far?

- A `pytest`
- B `pytest --cov`
- C `py -m pytest --cov`
- D `py -m pytest --no-cov`
- E `py -m pytest`
- F `py -m pytest --cov=imppkg`

Listing 5.1 Configuring coverage to measure branches

```
[coverage:run]
branch = True
```

With branch coverage enabled, possible branches are added to the statement count to determine total coverage. Run pytest again. Notice that the coverage for your code dropped from 67% to 50%, as shown next:

```
$ py -m pytest
...

-- coverage: platform darwin, python 3.10.0-beta-2 --
Name                    Stmts   Miss Branch BrPart  Cover
---------------------------------------------------------
.../__init__.py             0      0      0      0   100%
.../harmony.py              6      2      2      0    50%
---------------------------------------------------------
TOTAL                       6      2      2      0    50%
```

Two branches were found, and none were partially covered.

NOTE When branches are considered in coverage, the total coverage will be strictly less than or equal to the coverage without branching considered. The coverage percentage with branches considered can be difficult to calculate by hand because it considers all the different paths code may take during execution. You can read more about the specifics of branch measurement in the Coverage.py documentation (http://mng.bz/G1EA).

Now that you have a clearer picture of how well your tests cover your code and its execution paths, it's useful to know exactly which paths aren't covered.

Enabling missing coverage

Coverage.py can keep track of exactly which lines and branches aren't covered by tests, which is a big help as you try to write tests that increase the coverage of your code. You can turn this on by adding a new section to setup.cfg called `[coverage :report]`, with a new key called `show_missing` set to a value of `True` (see listing 5.2). This will produce one new `Missing` column in the coverage output. The `Missing` column lists the following:

- Lines or ranges of lines that aren't covered. As an example, `9` means line 9 is uncovered, and `10-12` means lines 10, 11, and 12 are uncovered.
- Logic flow from one line to another that represents a branch that isn't covered. As an example, `13->19` means the execution path that starts at line 13 that would next execute line 19 is uncovered.

Listing 5.2 Configuring coverage to show uncovered code

```
[coverage:report]
show_missing = True
```

Run pytest again to see what the coverage report says you're missing. The lines listed in the report will correspond to the lines of the `main` function body in the `harmony.py` module, as shown here:

```
$ py -m pytest
...

-- coverage: platform darwin, python 3.10.0-beta-2 --
Name              Stmts Miss Branch BrPart Cover Missing
---------------------------------------------------------
.../__init__.py       0    0      0      0  100%
.../harmony.py        6    2      2      0   50%    9-10
---------------------------------------------------------
TOTAL                 6    2      2      0   50%
```

Lines 9 and 10 are uncovered.

You can use the report of missing lines to quickly identify areas of focus for writing more tests.

Take a close look at the file paths in the Coverage.py output. They point to the files created in the virtual environment when you installed your package, with a prefix like .venv/lib/python3.10/site-packages/imppkg/. This is perfectly correct but can sometimes be difficult to read with the long prefix in front of each file. To simplify these paths and map the coverage back to the related source code, you can tell Coverage.py which file paths it should consider equivalent.

Simplifying coverage report output

In your project, the .venv/lib/python3.10/site-packages/imppkg/ directory of your installed package is roughly equivalent to the src/imppkg/ directory of the package's source code. Tell Coverage.py this is the case with a new section in setup.cfg called `[coverage:paths]`. Add a `source` key to this section, with a list value of equivalent file paths. Coverage.py will use the first entry to replace any subsequent entries in the output. Paths in this list can contain wildcard characters (`*`) to allow any name in that portion of the path to match. The new section should look like the next listing when you finish.

Listing 5.3 Configuring coverage to output paths related to the source code

```
[coverage:paths]
source =
    src/imppkg/
    */site-packages/imppkg/
```

Run pytest again. The file paths in the output will be prefixed with src/imppkg instead of .venv/lib/python3.10/site-packages/imppkg, as shown next:

```
$ py -m pytest
...
```

```
-- coverage: platform darwin, python 3.10.0-beta-2 --
Name                           Stmts Miss Branch BrPart Cover Missing
-----------------------------------------------------------------
src/imppkg/__init__.py             0    0      0      0  100%
src/imppkg/harmony.py              6    2      2      0   50%    9-10
-----------------------------------------------------------------
TOTAL                              6    2      2      0   50%
```

As your project grows and you spend more time testing, it might become harder to pick out uncovered modules from the coverage report. If you're reaching 100% coverage for several files, it can be helpful to ignore them in the report output. You can add a `skip_covered` key with a value of `True` to the `[coverage:report]` section to filter those out (see the next listing). Files that are filtered out are only removed from the list; their coverage is still considered in the total coverage calculation for your code.

Listing 5.4 Configuring coverage to skip covered files

```
[coverage:report]
...
skip_covered = True
```

Run pytest again. The `__init__.py` module will be filtered out of the report, with a message confirming that's the case, as follows:

```
$ py -m pytest
...

-- coverage: platform darwin, python 3.10.0-beta-2 --
Name                           Stmts Miss Branch BrPart Cover Missing
-----------------------------------------------------------------
src/imppkg/harmony.py              6    2      2      0   50%    9-10
-----------------------------------------------------------------
TOTAL                              6    2      2      0   50%

1 file skipped due to complete coverage.     <-- This confirms that fully
                                                  covered files are filtered out.
```

Now the coverage report shows you only those files that require your attention when your aim is to increase your test coverage.

5.1.3 Increasing test coverage

You've now got a trimmed-down way to see which files in your project may need testing attention with a report that can quickly let you know how changes you make impact the coverage. It's a great time to get a real test written to replace the `assert True` you wrote earlier.

In the `test_harmonic_mean.py` module, you need to write a test that exercises the code in the `harmony.py` module. The code there consists of the `main` function, which does the following:

1. Reads arguments from `sys.argv`
2. Converts those arguments to floating-point numbers
3. Calculates the harmonic mean of the numbers using the `harmonic_mean` function
4. Prints the result in colored text

You can write a test that will facilitate all these actions by patching `sys.argv` to a controlled value and asserting that the output is what you expect. This will also result in 100% coverage of the `harmony.py` module. However, this is what's known as a *happy path test.*

UNCOVERING UNHAPPY PATHS

Unhappy path tests exercise the less-frequent, error-prone ways through the code under test. When you want to make your code more robust, you should venture outside happy path tests to find these edge cases that may break your code (see figure 5.2).

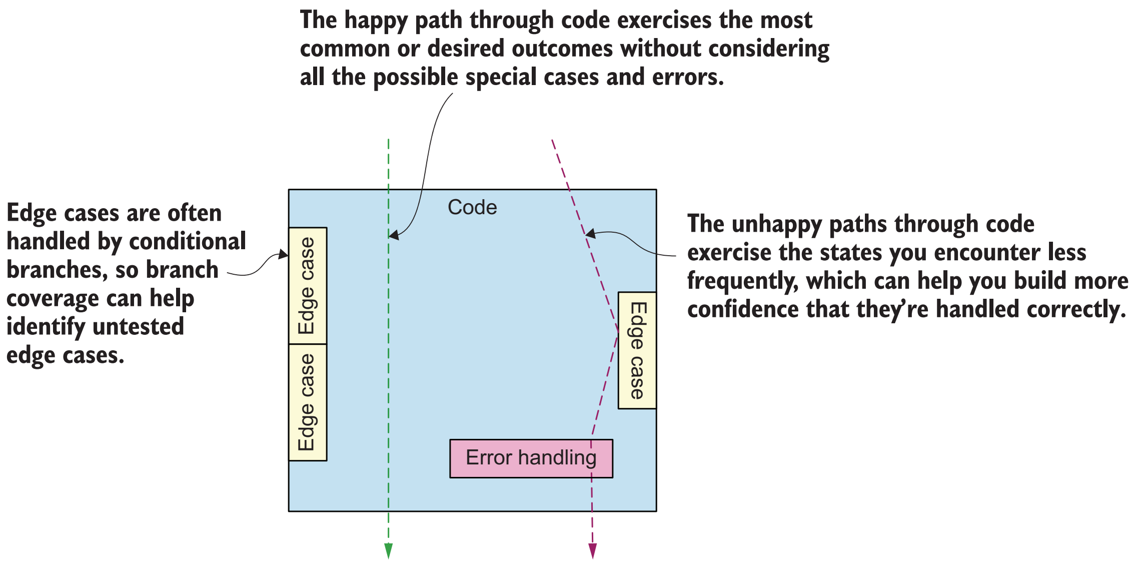
The happy path through code exercises the most common or desired outcomes without considering all the possible special cases and errors.
Edge cases are often handled by conditional branches, so branch coverage can help identify untested edge cases.
Code
Edge case
Edge case
Edge case
Error handling
The unhappy paths through code exercise the states you encounter less frequently, which can help you build more confidence that they're handled correctly.

Figure 5.2 Tests may cover the common, desired execution paths or the less common edge and error cases.

You might be wondering how you can write tests with 100% line *and* branch coverage that can still miss code failures. If you have tests for every execution path that all pass, how can there be a way for the code to fail? The reason often comes down to the inputs that a piece of code accepts, especially if that input can come directly from a user. In the case of the `harmony` console script, it accepts input directly from the user at the command line and passes it into the `harmony.py` module's `main` function. If that input is invalid, your code may handle it in an unexpected way. This serves as a good reminder that full test coverage still isn't a perfect protection against errors.

Try running the installed `harmony` command. Note that you'll need to run it using `.venv/bin/harmony` because you haven't installed your package globally and the harmony command isn't on your $PATH. What happens when you pass it arguments that can't be converted to numbers? What happens when you don't pass it any arguments at all? You can produce a ZeroDivisionError or a ValueError. So even though a happy path where you pass in numbers works correctly, it's still possible to produce undesired outcomes with carefully chosen inputs. It's up to you in these cases to choose between documenting the proper usage and ignoring the edge cases, or updating your code to accommodate.

For the moment, assume that any inputs that result in a division by zero or that can't be converted to numbers should result in an output of 0.0. One way to accommodate this in the code is with a try for each potentially dangerous operation and a catch to handle the corresponding exception (see listing 5.5). This can start to feel like *defensive programming*, where you guard against all possible risks, no matter how unlikely they might be. But for some applications, you want to provide an error-free outcome, either for user experience or safety. You want CarCorp to be happy, and with the back and forth you've already had with them, it seems worth covering your bases.

Listing 5.5 A safer version of the main function that handles poor inputs

```
def main():
    result = 0.0

    try:
        nums = [float(num) for num in sys.argv[1:]]
    except ValueError:
        nums = []

    try:
        result = harmonic_mean(nums)
    except ZeroDivisionError:
        pass

    cprint(result, 'red', 'on_cyan', attrs=['bold'])
```

The result will be zero unless successfully calculated later.

If any input can't be converted to a number, proceed as if there's no input.

If there's no input or the input is only zero, proceed with the default result.

This creates more lines and branches in the code, so you can expect the coverage to drop further. But now your coverage measurement can guide you to writing tests that assert the proper behavior for a wider variety of inputs. Update the source code in the harmony.py module to catch the ValueError and ZeroDivisionError cases. Then reinstall your package into your virtual environment using the py -m pip install . command.

Exercise 5.4

The following test covers the happy path of the main function, faking a user input and making an assertion about the printed output:

```
import sys

from termcolor import colored
```

(continued)

```
from imppkg.harmony import main


def test_harmony_happy_path(monkeypatch, capsys):
    inputs = ["1", "4", "4"]
    monkeypatch.setattr(sys, "argv", ["harmony"]
➥ + inputs)

    main()

    expected_value = 2.0
    assert capsys.readouterr().out.strip() == colored(
        expected_value,
        "red",
        "on_cyan",
        attrs=["bold"]
    )
```

pytest fixtures to set state and get command output

Values for which to calculate a harmonic mean, as strings

Passes the values as if they were supplied to harmony

Reads from sys.argv and performs the calculation

Asserts the output is 2.0 in colored text

Add this test to the `test_harmonic_mean.py` module and run pytest. You will see the coverage increase. What would you adjust for additional tests to cover the unhappy paths? How many additional tests do you need? Add a test for each of the unhappy paths you accounted for in the code to reach 100% coverage. Ensure that each test has a unique name so that pytest will run them all.

Earlier in this chapter, you configured Coverage.py to skip listing fully covered files. When you reach 100% coverage, all your files will disappear from the output because they're fully covered. The Coverage.py output shows an indication of 100% coverage as well, as shown next:

```
--- coverage: platform darwin, python 3.9.5-final-0 ---
Name     Stmts   Miss Branch BrPart  Cover   Missing
---------------------------------------------------
---------------------------------------------------
TOTAL       14      0      2      0   100%

2 files skipped due to complete coverage.
```

Indicates no missed statements or branches and 100% coverage

Indicates both files were skipped in the report due to full coverage

Now that you've reached 100% coverage, including some unhappy paths, you're in great shape. pytest will tell you if the behavior of your code regresses in the form of failing tests, and Coverage.py will tell you if there are any obvious opportunities for additional tests lurking. This leaves you free to get into the testing mindset and uncover unhappy paths that only you can identify. Now that you've got that out of the way, you'll take a few additional measures to further reduce the effort of testing going forward.

5.2 Addressing testing tedium

When you're new to testing, it can feel like a big thing standing in the way of getting things done. When you just want to deliver new features and value, tests can feel like a tangential effort. Reducing the effort of testing is a good way to encourage its adoption, and the investment will pay dividends in the future as your test suite grows.

5.2.1 Addressing repetitive, data-driven tests

You might have noticed that the tests you wrote to cover the `main` function all looked eerily similar. They each have the same basic shape, with just a few values altered. pytest has a great tool to address this kind of repeated, data-driven test. The `@pytest.mark.parametrize` decorator maps a list of values to arguments for the decorated test function, creating a separate test for each set of values. You can then use these arguments to construct a single test function that will properly assert the behavior for all the different values.

The `@pytest.mark.parametrize` decorator accepts the following arguments:

1. The argument names to map values to as a comma-separated string
2. A list, where each item is a tuple of values to map to the arguments

The decorated test function must accept arguments that correspond to the first `parametrize` argument, but it can accept additional arguments in any order. It's a common practice to place the parameterized arguments first and any additional arguments like fixtures last.

Imagine you've written a `mul` function that accepts two numeric arguments and returns their product. You want to write some tests that ensure it works properly when an input is positive, zero, and negative. You can use pytest's parameterization to do so, as shown in the following snippet:

```
import pytest

from ... import mul


@pytest.mark.parametrize(
    "input_one, input_two, expected",
    [
        (2, 3, 6),
        (-2, 3, -6),
        (-2, -3, 6),
        (0, 3, 0),
    ]
)
def test_mul(input_one, input_two, expected):
    assert mul(input_one, input_two) == expected
```

The argument names to which values are mapped

A list of tuples, each of which gets mapped

Argument names that match the specification to parametrize

A test constructed using the mapped arguments

This parameterized test function will result in four tests; each will have its own status in the pytest output. If one fails, the others can still pass. If you want to add more cases, it's a matter of adding a new tuple to the list of parameters. This can make it much faster to deal with data-heavy test suites with repetitive tests.

Exercise 5.5

Using `@pytest.mark.parametrize`, convert your tests for the `harmony.py` module's `main` function into a single, parameterized test. Don't forget to import `pytest`. After you finish, you should still have 100% coverage and the same number of passing tests.

Now that you've made your tests a bit leaner, you'll take a closer look at the testing process itself.

5.2.2 Addressing frequent package installation

You've installed your package into your virtual environment at least twice now. Because you set up your package to ensure you're always testing against the installed package, you need to reinstall the package each time you make any functional changes. This ensures that what you see matches what others see, but it also creates this manual work for you. You've only made one or two small changes to your source code so far, but imagine how you'll feel after your tenth feature request from CarCorp.

You also learned that making your package compatible with multiple dependencies and systems helps more people use it successfully. If you want to test your package with those different dependencies, that multiplies your manual work; each new dependency range causes further *combinatorial* growth (see figure 5.3). Combinatorial growth happens in a system where the number of possible states increases significantly with each new dimension added to the system. In your testing system, with only a few dependency variables, you can quickly reach tens of combinations to test.

tox (https://tox.readthedocs.io) automates the installation of packages for testing and the creation of a test matrix for dependency combinations. It significantly reduces the manual work you need to do, and as a result, it reduces the chance of human error in your testing.

GETTING STARTED WITH TOX

tox builds a fresh virtual environment for each combination of dependencies you test. Because of this isolated approach, you can make tox available globally and use it across projects instead of installing it separately in each.

NOTE If you haven't installed tox yet, head over to appendix B and return to this section when you're done.

From the root directory of your project, run the `tox` command. Because you haven't configured tox yet, you will see the following output:

```
$ tox

ERROR: tox config file (either pyproject.toml, tox.ini, setup.cfg) not found
```

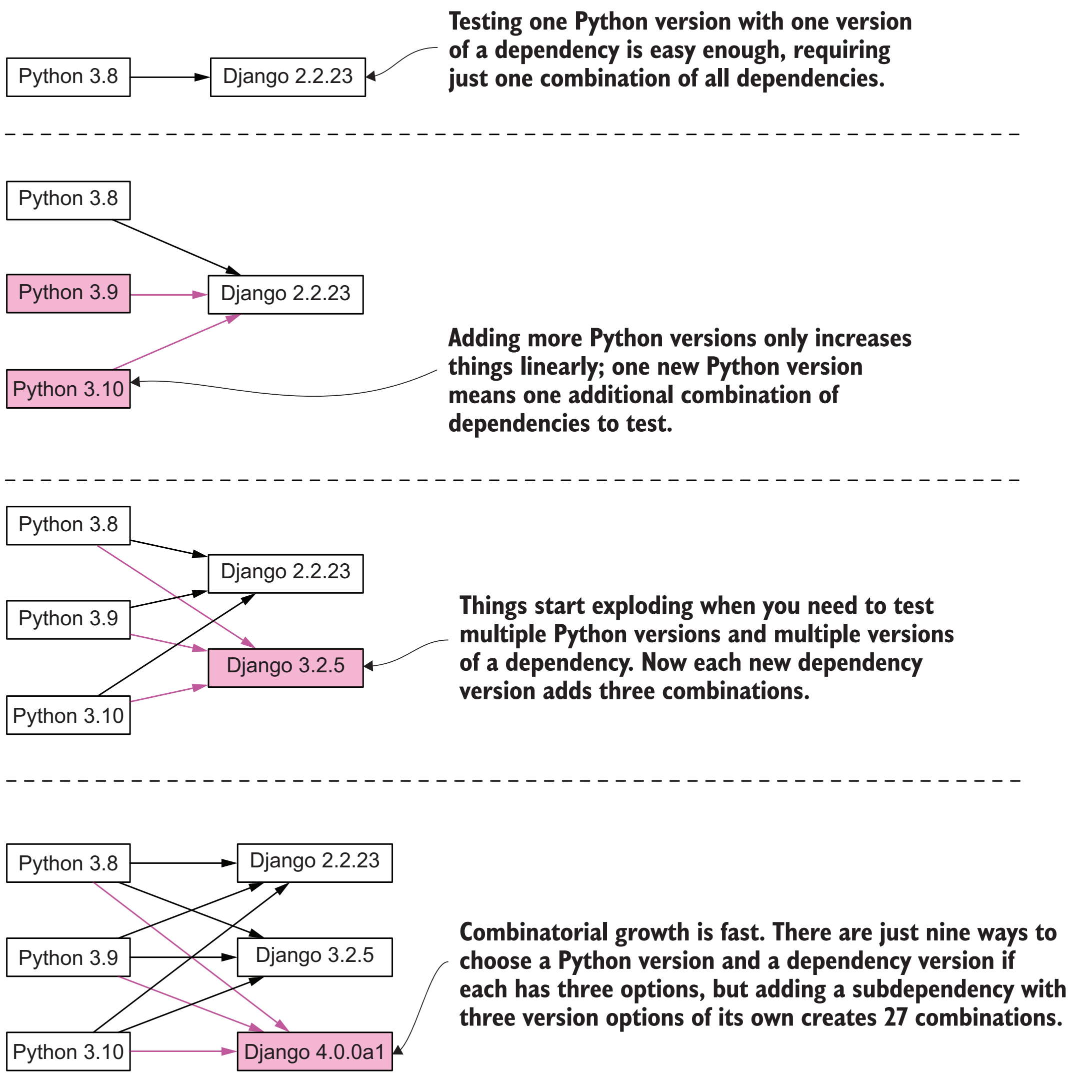
Python 3.8
Django 2.2.23
Testing one Python version with one version of a dependency is easy enough, requiring just one combination of all dependencies.
Python 3.8
Python 3.9
Python 3.10
Django 2.2.23
Adding more Python versions only increases things linearly; one new Python version means one additional combination of dependencies to test.
Python 3.8
Python 3.9
Python 3.10
Django 2.2.23
Django 3.2.5
Things start exploding when you need to test multiple Python versions and multiple versions of a dependency. Now each new dependency version adds three combinations.
Python 3.8
Python 3.9
Python 3.10
Django 2.2.23
Django 3.2.5
Django 4.0.0a1
Combinatorial growth is fast. There are just nine ways to choose a Python version and a dependency version if each has three options, but adding a subdependency with three version options of its own creates 27 combinations.

Figure 5.3 Testing across a range of versions for multiple dependencies grows very quickly.

Now add a new section to the setup.cfg file called `[tox:tox]`. This section is where you'll put the high-level configuration for your test matrix, as well as for tox itself. Start by adding an `isolated_build` key with a value of `True` as follows:

```
...

[tox:tox]
isolated_build = True
```

This tells tox to use the PEP 517 and PEP 518 standards you learned about in chapter 3 to build your package. Run tox again to confirm that it sees the configuration. tox produces the following friendly output:

```
$ tox

------------------
congratulations :)
```

With the confirmation that tox is reading your configuration, you're ready to start creating a test matrix.

The tox environment model

tox operates on the concept of *environments*. A tox environment is an isolated place to perform a set of commands, with its own set of installed dependencies and environment variables. Each tox environment includes a virtual environment with a copy of the Python interpreter (see figure 5.4). The tox configuration language gives you fine control over all of this, with a syntax that overcomes most of the challenges arising from the combinatorial nature of your test matrix.

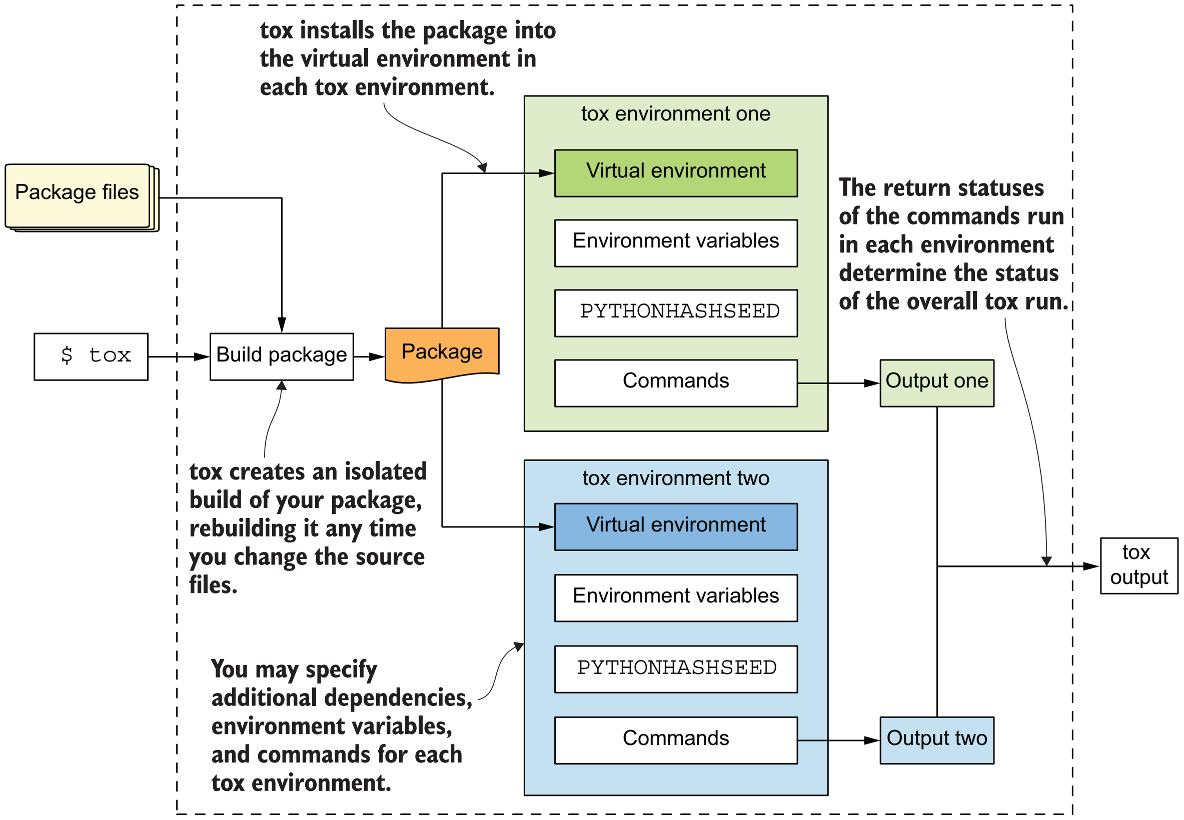
tox installs the package into the virtual environment in each tox environment.
Package files
$ tox
Build package
Package
tox environment one
Virtual environment
Environment variables
PYTHONHASHSEED
Commands
The return statuses of the commands run in each environment determine the status of the overall tox run.
Output one
tox creates an isolated build of your package, rebuilding it any time you change the source files.
tox environment two
Virtual environment
Environment variables
PYTHONHASHSEED
Commands
You may specify additional dependencies, environment variables, and commands for each tox environment.
Output two
tox output

Figure 5.4 tox environments are an isolated place to build, install, and test your code.

You can create any arbitrary environments that you wish, but tox treats a few environment names specially. Environments with names like `py37` or `py310` will create a virtual environment with a copy of the corresponding version of the Python interpreter.

The `envlist` key in the tox configuration defines which environments tox should create and execute by default when running the `tox` command. The environments in the `envlist` can also be run individually as desired by using the `-e` argument to the `tox` command and specifying the environment name.

To get started, add an `envlist` key to the `tox:tox` section in your setup.cfg file with a value of `py310` as follows:

```
[tox:tox]
...
envlist = py310
```

The list of tox environments is your test matrix.

The next time you run tox, it will

1. Create an isolated build of your package
2. Create a virtual environment with a copy of Python 3.10
3. Install your package in the virtual environment
4. Set `PYTHONHASHSEED` to a new value to create more randomness for tests

Run the `tox` command again. You will see output similar to the following:

```
.package create: .../first-python-package/.tox/.package
.package installdeps: setuptools, wheel, cython
py310 create: .../first-python-package/.tox/py310
py310 inst: .../first-python-package-0.0.1.tar.gz
py310 installed: first-python-package @
➥ file://.../first-python-package-0.0.1.tar.gz,
➥ termcolor==1.1.0
py310 run-test-pre: PYTHONHASHSEED='3663842017'
_____________________ summary _____________________
  py310: commands succeeded
  congratulations :)
```

An isolated package build using tox as the build frontend

The build backend dependencies

Virtual environment creation

Installation of your package

Successful installation of your package

Successful installation of package dependencies

Seeds Python's randomization

Any commands executed in the environment succeeded.

With only a small amount of configuration, tox is able to do all this work for you. You haven't told tox what commands to run in the environment yet, but your environment is there and ready. What if you want to do the same thing for multiple versions of Python? The `envlist` key accepts a comma-separated list of environments. As an example, you can specify `py39,py310` to create both a Python 3.9 and Python 3.10 environment.

Update your `envlist` value to include an environment for an additional Python version. Although you've specified a new environment to create, tox will skip building your package because it knows the source code hasn't changed since the last build. Similar to the `py310` environment you already created, in the `py39` environment, tox will

1. Create the virtual environment
2. Install your package into it
3. Set PYTHONHASHSEED

tox will then execute the py310 environment again. Because it already existed, tox won't recreate it or reinstall dependencies unless it detects that the dependencies have changed. Run tox again. You will see output similar to the following:

```
py39 create: .../first-python-package/.tox/py39
py39 inst: .../first-python-package-0.0.1.tar.gz
py39 installed: first-python-package @
➥ file://.../first-python-package-0.0.1.tar.gz,
➥ termcolor==1.1.0
py39 run-test-pre: PYTHONHASHSEED='973215353'
py310 inst-nodeps: .../first-python-package-0.0.1.tar.gz
py310 installed: first-python-package @
➥ file://.../first-python-package-0.0.1.tar.gz,
➥ termcolor==1.1.0
py310 run-test-pre: PYTHONHASHSEED='973215353'
____________________ summary ____________________
  py39: commands succeeded
  py310: commands succeeded
  congratulations :)
```

The py39 environment is added.

The inst-nodeps skips installing dependencies.

Confirmation of each environment executed

You doubled the size of your test matrix by adding a few characters to the tox configuration. This becomes more and more valuable as you expand the combinations of dependencies you need to test, because you don't need to specify the combinations individually. tox will also ensure that your tests execute for each combination, which maximizes the chance of uncovering a bug specific to a given combination.

Because adding a new combination of dependencies will execute the tests an additional time, the total execution time of your test suite will grow. It can be helpful to run your tests in a single environment using the -e option while you're changing your code or your tests, and then run tox without specifying an argument after you've made your changes to ensure nothing has broken across all environments. You can also run multiple environments in parallel, which is covered later in this chapter.

Now you have two testing environments, but neither one does anything yet. The next step is to tell tox what to do within each environment.

5.2.3 Configuring test environments

So far you've configured tox in the [tox:tox] section to indicate how to build your package and which environments to create. To configure the test environments themselves, add a new [testenv] section. This section is used by default for any configured test environment. In this section, you tell tox what commands to run using the commands key. This key accepts a list of commands to run, with some special syntax available to pass arguments to the commands.

Within each command, you can use the {posargs} placeholder, which will pass any arguments to specify to the tox command along to the test environment commands.

As an example, if you specify `python -c 'print("{posargs}")'` as a command, running `tox hello world` will execute `python -c 'print("hello world")'` in the environment.

You can also pass options to a test command by separating them from the `tox` command and any of its options with two dashes (`--`). As an example, if you specify `python` as a command, running `tox -- -V` will execute `python -V` in the environment.

Exercise 5.6

Your test environments should execute the `pytest` command, with the ability to pass it additional arguments when running tox. Which of the following show a valid test command and corresponding tox command?

- A `pytest {posargs}`, `tox`
- B `pytest {posargs}`, `tox --no-cov`
- C `pytest {posargs}`, `tox -- --no-cov`
- D `pytest --no-cov {posargs}`, `tox`
- E `pytest {posargs} --no-cov`, `tox`
- F `{posargs} pytest`, `tox -- --no-cov`

After you add the `pytest` command to the `commands` list, run tox again. You'll see that after the steps you saw previously, tox tries to execute pytest and fails, as shown in the following output:

```
py39 run-test: commands[0] | pytest
ERROR: InvocationError for command
➥ could not find executable pytest
```

Attempted execution of the correct command

The command can't be found in the test environment.

Even though you installed pytest into the virtual environment for your project earlier, recall that tox creates and uses an isolated virtual environment for each test environment. This means that tox won't use the copy of pytest that you've been running. You haven't told tox to install pytest in those environments, so it can't find a copy there either. You can specify dependencies in the `[testenv]` section using the `deps` key. The value for `deps` is a list of Python packages to install, with syntax similar to `requirements.txt` or `install_requires`. For now, add `pytest` and `pytest-cov` as dependencies like so:

```
[testenv]
...
deps =
    pytest
    pytest-cov
```

Run tox again. This time it will install the additional dependencies, and the `pytest` command will successfully run the tests and the coverage report, with output similar to the following:

```
...
py39 installdeps: pytest, pytest-cov
...
py39 run-test: commands[0] | pytest
<PYTEST OUTPUT>
<COVERAGE OUTPUT>
...
py310 installdeps: pytest, pytest-cov
...
py310 run-test: commands[0] | pytest
<PYTEST OUTPUT>
<COVERAGE OUTPUT>
____________________ summary ____________________
  py39: commands succeeded
  py310: commands succeeded
  congratulations :)
```

You've now got pytest and coverage running successfully in isolated environments on two different Python versions, without having to install your package manually. Any time you make a change to your source code, dependencies, or tests, you can run tox to see if things are still working. This early investment in infrastructure—especially for those who prefer test-driven development—will pay dividends throughout the rest of a package's life.

Before you move on, read the next section for some additional testing and configuration tips.

5.2.4 Tips for quicker and safer testing

As your project grows, you run the risk that the time you spend testing will grow along with it. To keep your productivity up, you want to keep testing as fast as possible and to reduce human error as much as possible. The following sections discuss some tips to keep your test suite execution in check.

RUNNING TEST ENVIRONMENTS IN PARALLEL

You might have noticed that your Python 3.9 and Python 3.10 environments have been executing sequentially. They each take a few seconds, so it's not too big of a deal. Now imagine a project where you're testing three Python versions and three different versions of a dependency. Are you patient enough to wait for nine environments to run sequentially?

tox provides a parallel mode (http://mng.bz/z546) that can execute multiple environments at a time. To run your two environments in parallel automatically, pass the `-p` option when running tox, as shown in the next code snippet. This mode will hide the output from each individual environment by default, showing only a progress indicator for the active environments and an overall pass or fail status for each environment:

```
$ tox -p
⠹ [2] py39 | py310

...
```

```
✓ OK py39 in 9.533 seconds
✓ OK py310 in 9.96 seconds
__________ summary __________
  py39: commands succeeded
  py310: commands succeeded
  congratulations :)
```

Uncovering stateful tests

Consider the following snippet with two tests that make assertions about how Python lists work:

```
FRUITS = ["apple"]

def test_len():
    assert len(FRUITS) == 1

def test_append():
    FRUITS.append("banana")
    assert FRUITS == ["apple", "banana"]
```

Can you spot the issue? It might be subtle, but the second test alters the state of the system. Although `FRUITS` starts out containing one item, `"apple"`, the test alters the list by adding `"banana"`. These tests will pass as written, but they'll fail if you put them in reverse order (see figure 5.5):

```
FRUITS = ["apple"]

def test_append():
    FRUITS.append("banana")
    assert FRUITS == ["apple", "banana"]

def test_len():
    assert len(FRUITS) == 1
```

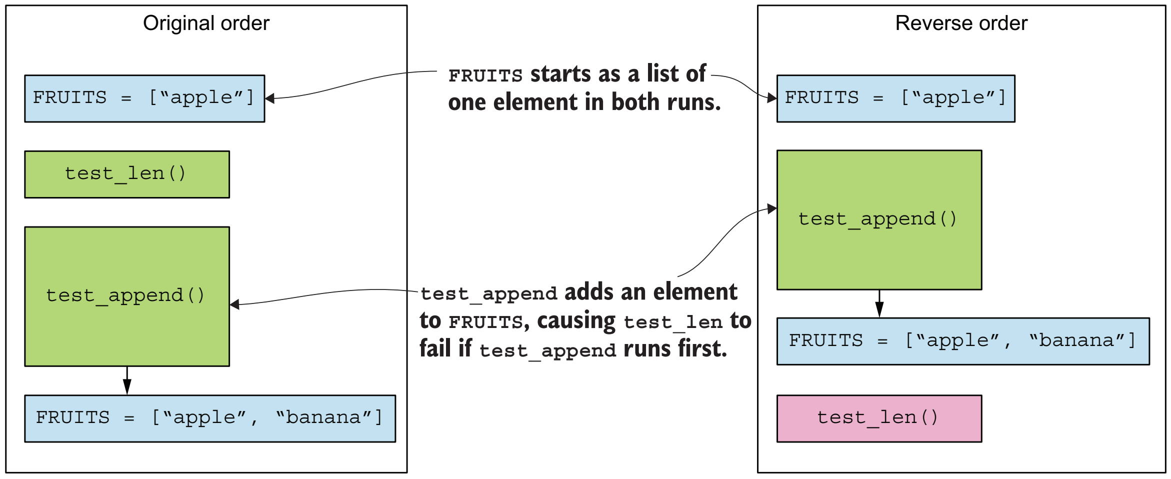
Original order
FRUITS = ["apple"]
test_len()
test_append()
FRUITS = ["apple", "banana"]
FRUITS starts as a list of one element in both runs.
test_append adds an element to FRUITS, causing test_len to fail if test_append runs first.
Reverse order
FRUITS = ["apple"]
test_append()
FRUITS = ["apple", "banana"]
test_len()

Figure 5.5 Tests that depend on state created by other tests may fail when reordered or moved.

Although it might be easy enough to spot and fix this example, stateful tests are often the result of several layers and interactions that you might not notice while writing the code. To increase the likelihood that you'll find and uncover these situations, you should run your tests in a random order. The `pytest-randomly` plugin (https://github.com/pytest-dev/pytest-randomly) does exactly this. It requires no configuration for the basic behavior of randomly ordered tests; add it to your `[testenv]` section's `deps` list, and you'll be all set.

By swapping the order of test modules, classes, methods, and functions, `pytest-randomly` uncovers tests that fail because of a dependency on state created in an earlier test (see figure 5.6). It does this by changing the random seed to a repeatable value for each run. This information is added to the pytest output shown here:

```
Using --randomly-seed=1966324489
```

When a test run produces a test failure, you can force future runs to execute in the same order that produced the failure by passing the `--randomly-seed` option to the `pytest` command with the same value output in the original run. Because pytest is being run by tox, you can pass the option to the underlying `pytest` command using a `--` to separate tox options from pytest options like this:

```
$ tox -- --randomly-seed=1966324489
```

tox passes the argument to pytest.

With `pytest-randomly` installed, your tests will run in a different order each time you run tox. If you notice that a test occasionally fails for no obvious reason, the test or the code under test may be stateful. You can use these hints as a good starting place to hunt down stateful issues.

ENSURING PYTEST MARKERS ARE VALID

Earlier in this chapter, you used the `@pytest.mark.parametrize` marker to parameterize a data-driven test. Although pytest provides built-in markers like `parametrize`, you can also devise your own arbitrary markers; in a way, you can think of them as labels or tags for your tests. This is a powerful feature, but because you can create arbitrary markers, there's a chance you'll misremember or misspell the name of a marker, which could cause silent issues down the road.

By default, pytest will gently warn you about an invalid marker, as you can see in the following:

```
... PytestUnknownMarkWarning: Unknown pytest.mark.fake - is this a typo?
```

If you want to ensure that all your markers are known and valid—that is, they're registered by a plugin or in your `[tool:pytest]` section in the `markers` key—add the `--strict-markers` option to the `addopts` key in the setup.cfg file. With strict markers enabled, pytest will fail a test run if it finds an unknown marker, as you can see in the following output:

```
'fake' not found in `markers` configuration option
```

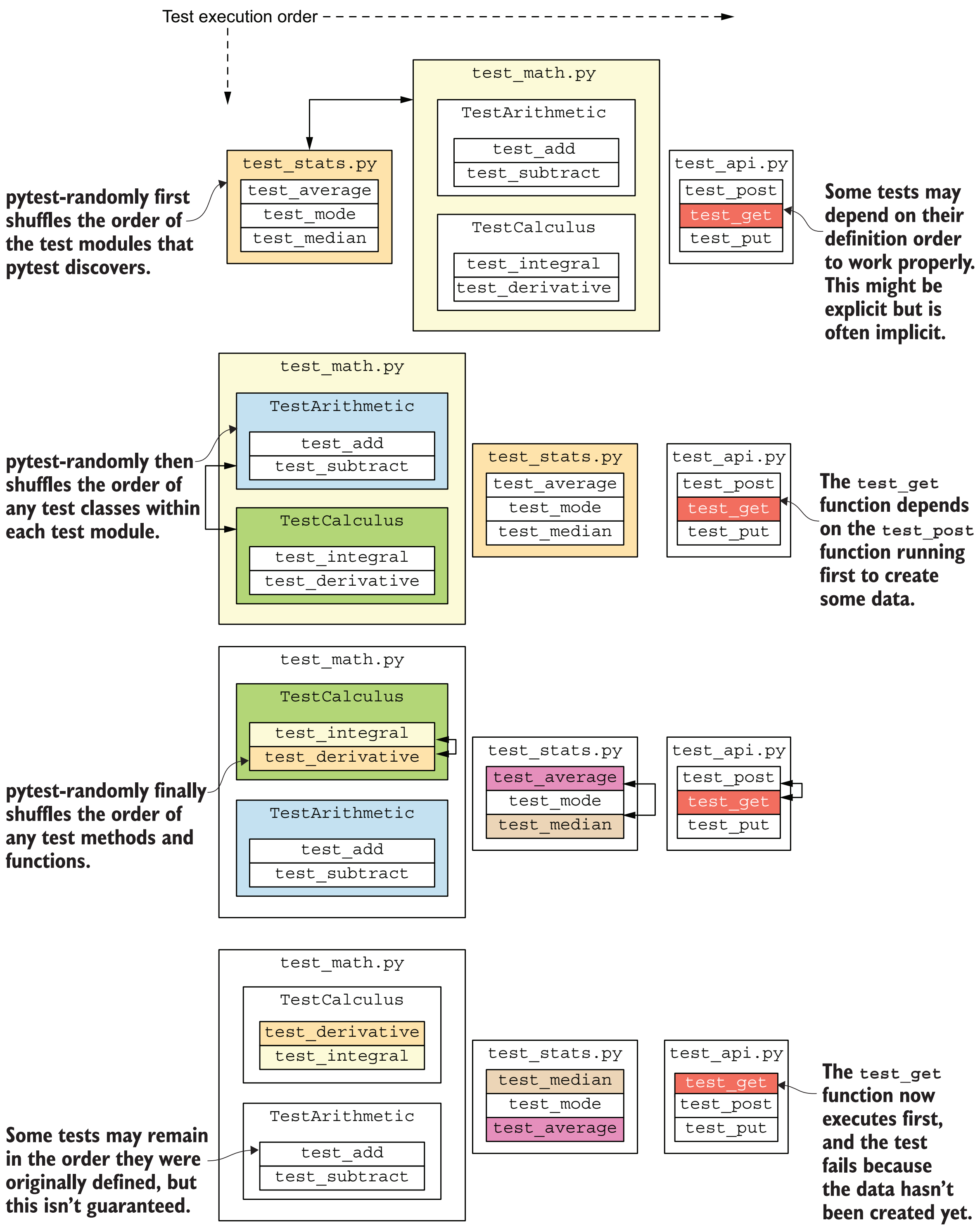
Test execution order
test_math.py
TestArithmetic
test_add
test_subtract
TestCalculus
test_integral
test_derivative
test_stats.py
test_average
test_mode
test_median
test_api.py
test_post
test_get
test_put
pytest-randomly first shuffles the order of the test modules that pytest discovers.
Some tests may depend on their definition order to work properly. This might be explicit but is often implicit.
pytest-randomly then shuffles the order of any test classes within each test module.
The test_get function depends on the test_post function running first to create some data.
pytest-randomly finally shuffles the order of any test methods and functions.
Some tests may remain in the order they were originally defined, but this isn't guaranteed.
The test_get function now executes first, and the test fails because the data hasn't been created yet.

Figure 5.6 `pytest-randomly` runs your tests in a shuffled order on each run.

This will ensure that your tests run only if you have a valid set of markers defined. An invalid marker isn't harmful by itself, but it minimizes the chance of unexpected behavior.

ENSURING EXPECTED FAILURES DON'T PASS UNEXPECTEDLY

pytest provides a marker called `xfail` that marks a test as an expected failure. A test might be expected to fail for a variety of reasons—environmental issues, an upstream issue you're waiting on, or simply a lack of time to adress it. Occasionally, an expected failure can start passing again after you make a change. It might sound good to have more passing tests, but an unexpected change in behavior should always engender some scrutiny.

By default, pytest will warn you about this situation by marking a test as `XPASS`. If you want to be loudly alerted to this situation so that you can examine why an expected failure started passing, add the `xfail_strict` key to the `[tool:pytest]` section with a value of `True`. This will cause any passing tests that were expected to fail to fail the test run so that you have to address them before continuing.

With your lean, mean, testing machine well-oiled and ready for any changes you throw at it, you're ready to start adding and automating more code quality processes in the next chapter.

Answers to exercises

5.3—Answer: E

5.4—Answer: Add two new tests that adjust `inputs` to `[]` and, for example, `["foo", "bar"]` respectively, and adjust `expected_value` to `0.0` for both.

5.5

```
@pytest.mark.parametrize(
    "inputs, expected",
    [
        (["1", "4", "4"], 2.0),
        ([], 0.0),
        (['foo', 'bar'], 0.0),
    ]
)
def test_harmony_parametrized(inputs, expected, monkeypatch, capsys):
    monkeypatch.setattr(sys, 'argv', ['harmony'] + inputs)
    main()
    assert capsys.readouterr().out.strip() == colored(
        expected,
        'red',
        'on_cyan',
        attrs=['bold']
    )
```

5.6—Answer: A, C, D, E

B would pass the `--no-cov` option to tox itself instead of pytest. F would put any passed arguments before the command.

Summary

- The pytest framework has a rich plugin ecosystem that you can use to test more productively than using the built-in `unittest` module.
- Use test coverage to guide the tests you write by identifying areas of code that aren't executed by your existing tests.
- Test the uncommon paths through your code because coverage is useful, but not sufficient to understand how well your tests ensure the proper behavior.
- Testing many combinations of dependencies is tedious and error-prone, but tox reduces this effort and increases safety by automating most of the steps involved.
- To maximize safety, use plugins and tool options to your advantage to restrict your project to only valid configurations.