Extracted from:

Python Testing with pytest, Second Edition

Simple, Rapid, Effective, and Scalable

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Python Testing with pytest

Second Edition

Simple, Rapid, Effective, and Scalable

Brian Okken

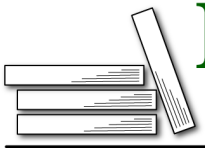
edited by Katharine Dvorak

Python Testing with pytest, Second Edition

Simple, Rapid, Effective, and Scalable

Brian Okken

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: Katharine Dvorak
Copy Editor: Karen Galle
Indexing: Potomac Indexing, LLC
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: 978-1-68050-860-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—February 2022

CHAPTER 6

Markers

In pytest, *markers* are a way to tell pytest there's something special about a particular test. You can think of them like tags or labels. If some tests are slow, you can mark them with @pytest.mark.slow and have pytest skip those tests when you're in a hurry. You can pick a handful of tests out of a test suite and mark them with @pytest.mark.smoke and run those as the first stage of a testing pipeline in a continuous integration system. Really, for any reason you might have for separating out some tests, you can use markers.

pytest includes a handful of builtin markers that modify the behavior of how tests are run. We've used one already, @pytest.mark.parametrize, in Parametrizing Functions, on page ?. In addition to the custom tag-like markers we can create and add to our tests, the builtin markers tell pytest to do something special with the marked tests.

In this chapter, we're going to explore both types of markers: the builtins that change behavior, and the custom markers we can create to select which tests to run. We can also use markers to pass information to a fixture used by a test. We'll take a look at that, too.

Using Builtin Markers

pytest's builtin markers are used to modify the behavior of how tests run. We explored @pytest.mark.parametrize() in the last chapter. Here's the full list of the builtin markers included in pytest as of pytest 6:

- @pytest.mark.filterwarnings(warning): This marker adds a warning filter to the given test.
- @pytest.mark.skip(reason=None): This marker skips the test with an optional reason.

- @pytest.mark.skipif(condition, ..., *, reason): This marker skips the test if any of the conditions are True.
- @pytest.mark.xfail(condition, ..., *, reason, run=True, raises=None, strict=xfail_strict): This marker tells pytest that we expect the test to fail.
- @pytest.mark.parametrize(argnames, argvalues, indirect, ids, scope): This marker calls a test function multiple times, passing in different arguments in turn.
- @pytest.mark.usefixtures(fixturename1, fixturename2, ...): This marker marks tests as needing all the specified fixtures.

These are the most commonly used of these builtins:

- @pytest.mark.parametrize()
- @pytest.mark.skip()
- @pytest.mark.skipif()
- @pytest.mark.xfail()

We used parametrize() in the last chapter. Let's go over the other three with some examples to see how they work.

Skipping Tests with pytest.mark.skip

The skip marker allows us to skip a test. Let's say we're thinking of adding the ability to sort in a future version of the Cards application, so we'd like to have the Card class support comparisons. We write a test for comparing Card objects with < like this:

ch6/builtins/test_less_than.py

```
from cards import Card

def test_less_than():
    c1 = Card("a task")
    c2 = Card("b task")
    assert c1 < c2

def test_equality():
    c1 = Card("a task")
    c2 = Card("a task")
    assert c1 == c2
```

And it fails:

```
$ cd /path/to/code/ch6/builtins
$ pytest --tb=short test_less_than.py
========================= test session starts ==========================
collected 2 items

test_less_than.py F.                                             [100%]
```

```
============================== FAILURES ==============================
_____________________________ test_less_than _____________________________
test_less_than.py:6: in test_less_than
    assert c1 < c2
E   TypeError: '<' not supported between instances of 'Card' and 'Card'
======================= short test summary info ========================
FAILED test_less_than.py::test_less_than - TypeError: '<' not support...
====================== 1 failed, 1 passed in 0.13s ======================
```

Now the failure isn't a shortfall of the software; it's just that we haven't finished this feature yet. So what do we do with this test?

One option is to skip it. Let's do that:

ch6/builtins/test_skip.py

```
import pytest

@pytest.mark.skip(reason="Card doesn't support < comparison yet")
def test_less_than():
    c1 = Card("a task")
    c2 = Card("b task")
    assert c1 < c2
```

The @pytest.mark.skip() marker tells pytest to skip the test. The reason is optional, but it's important to list a reason to help with maintenance later.

When we run skipped tests, they show up as s:

```
$ pytest test_skip.py
========================== test session starts ==========================
collected 2 items

test_skip.py s.                                                    [100%]

====================== 1 passed, 1 skipped in 0.03s ======================
```

Or as SKIPPED in verbose:

```
$ pytest -v -ra test_skip.py
========================== test session starts ==========================
collected 2 items

test_skip.py::test_less_than SKIPPED (Card doesn't support <...) [ 50%]
test_skip.py::test_equality PASSED                                 [100%]

======================= short test summary info ========================
SKIPPED [1] test_skip.py:6: Card doesn't support < comparison yet
====================== 1 passed, 1 skipped in 0.03s ======================
```

The extra line at the bottom lists the reason we gave in the marker, and is there because we used the -ra flag in the command line. The -r flag tells pytest to report reasons for different test results at the end of the session. You give it a single character that represents the kind of result you want more

information on. The default display is the same as passing in -rfE: f for failed tests; E for errors. You can see the whole list with pytest --help.

The a in -ra stands for "all except passed." The -ra flag is therefore the most useful, as we almost always want to know the reason why certain tests did not pass.

We can also be more specific and only skip the test if certain conditions are met. Let's look at that next.

Skipping Tests Conditionally with pytest.mark.skipif

Let's say we know we won't support sorting in the 1.x.x versions of the Cards application, but will in version 2.x.x. We can tell pytest to skip the test for all versions of Cards lower than than 2.x.x like this:

ch6/builtins/test_skipif.py

```
import cards
from packaging.version import parse

@pytest.mark.skipif(
    parse(cards.__version__).major < 2,
    reason="Card < comparison not supported in 1.x",
)
def test_less_than():
    c1 = Card("a task")
    c2 = Card("b task")
    assert c1 < c2
```

The skipif marker allows you to pass in as many conditions as you want and if any of them are true, the test is skipped. In our case, we are using packaging.version.parse to allow us to isolate the major version and compare it against the number 2.

This example uses a third-party package called packaging. If you want to try the example, pip install packaging first. version.parse is just one of the many handy utilities found there. See the packaging documentation[1] for more information.

With both the skip and the skipif markers, the test is not actually run. If we want to run the test anyway, we can use xfail.

Another reason we might want to use skipif is if we have tests that need to be written differently on different operating systems. We can write separate tests for each OS and skip on the inappropriate OS.

1. https://packaging.pypa.io/en/latest/version.html

Expecting Tests to Fail with pytest.mark.xfail

If we want to run all tests, even those that we know will fail, we can use the xfail marker.

Here's the full signature for xfail:

```
@pytest.mark.xfail(condition, ..., *, reason, run=True,
raises=None, strict=xfail_strict)
```

The first set of parameters to this fixture are the same as skipif. The test is run anyway, by default, but the run parameter can be used to tell pytest to not run the test by setting run=False. The raises parameter allows you to provide an exception type or a tuple of exception types that you want to result in an xfail. Any other exception will cause the test to fail. strict tells pytest if passing tests should be marked as XPASS (strict=False) or FAIL, strict=True.

Let's look at an example:

ch6/builtins/test_xfail.py

```
@pytest.mark.xfail(
    parse(cards.__version__).major < 2,
    reason="Card < comparison not supported in 1.x",
)
def test_less_than():
    c1 = Card("a task")
    c2 = Card("b task")
    assert c1 < c2

@pytest.mark.xfail(reason="XPASS demo")
def test_xpass():
    c1 = Card("a task")
    c2 = Card("a task")
    assert c1 == c2

@pytest.mark.xfail(reason="strict demo", strict=True)
def test_xfail_strict():
    c1 = Card("a task")
    c2 = Card("a task")
    assert c1 == c2
```

We have three tests here: one we know will fail and two we know will pass. These tests demonstrate both the failure and passing cases of using xfail and the effect of using strict. The first example also uses the optional condition parameter, which works like the conditions of skipif.

Here's what they look like when run:

```
$ pytest -v -ra test_xfail.py
========================= test session starts ==========================
collected 3 items

test_xfail.py::test_less_than XFAIL (Card < comparison not s...) [ 33%]
test_xfail.py::test_xpass XPASS (XPASS demo)                     [ 66%]
test_xfail.py::test_xfail_strict FAILED                          [100%]

============================== FAILURES ================================
_____________________________ test_xfail_strict _____________________________
[XPASS(strict)] strict demo
======================== short test summary info =======================
XFAIL test_xfail.py::test_less_than
  Card < comparison not supported in 1.x
XPASS test_xfail.py::test_xpass XPASS demo
FAILED test_xfail.py::test_xfail_strict
=============== 1 failed, 1 xfailed, 1 xpassed in 0.11s ================
```

For tests marked with xfail:

- Failing tests will result in XFAIL.
- Passing tests (with no strict setting) will result in XPASSED.
- Passing tests with strict=true will result in FAILED.

When a test fails that is marked with xfail, pytest knows exactly what to tell you: "You were right, it did fail," which is what it's saying with XFAIL. For tests marked with xfail that actually pass, pytest is not quite sure what to tell you. It could result in XPASSED, which roughly means, "Good news, the test you thought would fail just passed." Or it could result in FAILED, or, "You thought it would fail, but it didn't. You were wrong."

So you have to decide. Should your passing xfail tests result in XFAIL? If yes, leave strict alone. If you want them to be FAILED, then set strict. You can either set strict as an option to the xfail marker like we did in this example, or you can set it globally with the xfail_strict=true setting in pytest.ini, which is the main configuration file for pytest.

A pragmatic reason to always use xfail_strict is because we tend to look closely at all failed tests. Setting strict makes you look into the the cases where your test expectations don't match the code behavior.

There are a couple additional reasons why you might want to use xfail:

- You're writing tests first, test-driven development style, and are in the test writing zone, writing a bunch of test cases you know aren't implemented yet but that you plan on implementing shortly. You can mark the new behaviors with xfail and remove the xfail gradually as you implement the

behavior. This is really my favorite use of xfail. Try to keep the xfail tests on the feature branch where the feature is being implemented.

Or

- Something breaks, a test (or more) fails, and the person or team that needs to fix the break can't work on it right away. Marking the tests as xfail, strict=true, with the reason written to include the defect/issue report ID is a decent way to keep the test running, not forget about it, and alert you when the bug is fixed.

There are also bad reasons to use use xfail or skip. Here's one:

Suppose you're just brainstorming behaviors you may or may not want in future versions. You can mark the tests as xfail or skip just to keep them around for when you do want to implement the feature. Um, no.

In this case, or similar, try to remember YAGNI ("Ya Aren't Gonna Need It"), which comes from Extreme Programming and states: "Always implement things when you actually need them, never when you just foresee that you need them."[2] It can be fun and useful to peek ahead and write tests for bits of functionality you are just about to implement. However, it's a waste of time to try to look too far into the future. Don't do it. Our ultimate goal is to have all tests pass, and skip and xfail are not passing.

The builtin markers skip, skipif, and xfail are quite handy when you need them, but can quickly become overused. Just be careful.

Now let's switch gears and look at markers that we create ourselves to mark tests we want to run or skip as a group.

2. http://c2.com/xp/YouArentGonnaNeedIt.html