

Multiply your Testing Effectiveness with Parametrized Testing

Brian Okken

 [@brianokken](https://twitter.com/brianokken)

Code and slides

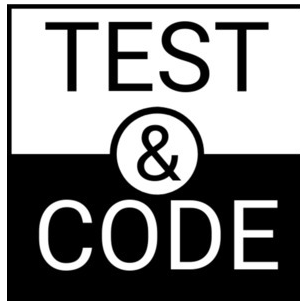
github.com/okken/talks

Brian Okken

Work



Podcasts



Book



 @brianokken

Value of Tests

A passing test suite

- Gives you confidence in what you are building
 - I didn't break anything that used to work
 - Future changes won't break current features
- Allows you to have pride in your work
 - I can refactor until I'm proud of the code
- Lets you play with the code and change it with less fear
- Helps build team trust
 - Code reviews can focus on team understanding and ownership

Only works if:

- New features are tested with new tests
- Everyone can read the tests
- **Tests are easy and fast to write** <- *this is what we're focusing on*

Takeaways

- Why parametrization is useful
- Your choices
 - function
 - fixture
 - `pytest_generate_tests`
- How to
 - run subsets of test cases
 - use `pytest.param` for ids and markers
 - use `indirect` to intercept parameters with fixtures

Parametrize vs Parameterize

parameter + ize

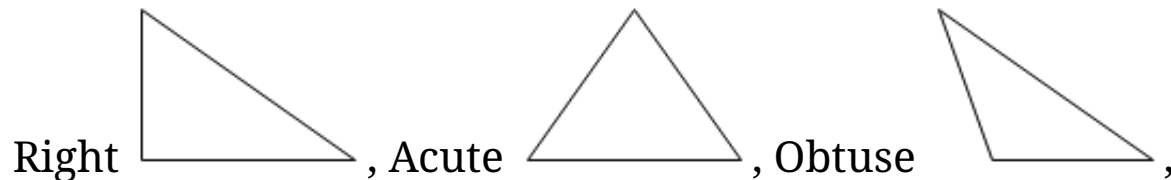
- *parameterize* (US)
- *parametrize* (UK)

pytest uses *parametrize*, the UK spelling.

Something to Test

triangles.py:

```
def triangle_type(a, b, c):  
    """  
    Given three angles,  
    return 'right', 'obtuse', 'acute', or 'invalid'.  
    """  
    angles = (a, b, c)  
    if 90 in angles:  
        return "right"  
    if any([a > 90 for a in angles]):  
        return "obtuse"  
    if all([a < 90 for a in angles]):  
        return "acute"  
    if sum(angles) != 180:  
        return "invalid"
```



pytest.ini

I wanted all the examples to include:

- `--tb=no` for "no tracebacks" to hide tracebacks
- `-v` for verbose to show the test names

So those are in a `pytest.ini` file:

```
[pytest]
addopts = --tb=no -v
markers =
    smoke : smoke tests
```

We're going to use the "smoke" marker later, so it's registered here also.

without Parametrization

```
def test_right():  
    assert triangle_type(90, 60, 30) == "right"  
  
def test_obtuse():  
    assert triangle_type(100, 40, 40) == "obtuse"  
  
def test_acute():  
    assert triangle_type(60, 60, 60) == "acute"  
  
def test_invalid():  
    assert triangle_type(0, 0, 0) == "invalid"
```

```
$ pytest test_1.py  
===== test session starts =====  
  
test_1.py::test_right PASSED [ 25%]  
test_1.py::test_obtuse PASSED [ 50%]  
test_1.py::test_acute PASSED [ 75%]  
test_1.py::test_invalid FAILED [100%]  
  
===== 1 failed, 3 passed in 0.03s =====
```


Moving to one test (don't do this)

```
def test_type():  
    many_triangles = [  
        (90, 60, 30, "right"),  
        (100, 40, 40, "obtuse"),  
        (60, 60, 60, "acute"),  
        (0, 0, 0, "invalid"),  
    ]  
    for a, b, c, expected in many_triangles:  
        assert triangle_type(a, b, c) == expected
```

```
$ pytest test_2.py  
===== test session starts =====  
  
test_2.py::test_type FAILED [100%]  
  
===== 1 failed in 0.03s =====
```

Function Parametrization

```
@pytest.mark.parametrize( 'a, b, c, expected', [  
    (90, 60, 30, "right"),  
    (100, 40, 40, "obtuse"),  
    (60, 60, 60, "acute"),  
    (0, 0, 0, "invalid")])  
  
def test_func(a, b, c, expected):  
    assert triangle_type(a, b, c) == expected
```

Function Parametrization

```
@pytest.mark.parametrize( 'a, b, c, expected', [
    (90, 60, 30, "right"),
    (100, 40, 40, "obtuse"),
    (60, 60, 60, "acute"),
    (0, 0, 0, "invalid")]
)
def test_func(a, b, c, expected):
    assert triangle_type(a, b, c) == expected
```

```
$ pytest test_3.py
===== test session starts =====

test_3.py::test_func[90-60-30-right] PASSED          [ 25%]
test_3.py::test_func[100-40-40-obtuse] PASSED         [ 50%]
test_3.py::test_func[60-60-60-acute] PASSED           [ 75%]
test_3.py::test_func[0-0-0-invalid] FAILED            [100%]

===== 1 failed, 3 passed in 0.03s =====
```

Parameters from a named list

```
many_triangles = [  
    (90, 60, 30, "right"),  
    (100, 40, 40, "obtuse"),  
    (60, 60, 60, "acute"),  
    (0, 0, 0, "invalid")  
]  
  
@pytest.mark.parametrize( 'a, b, c, expected', many_triangles)  
def test_func(a, b, c, expected):  
    assert triangle_type(a, b, c) == expected
```

Parameters from a function

```
def many_triangles():  
    return [ (90, 60, 30, "right"),  
             (100, 40, 40, "obtuse"),  
             (60, 60, 60, "acute"),  
             (0, 0, 0, "invalid") ]  
  
@pytest.mark.parametrize( 'a, b, c, expected', many_triangles())  
def test_func(a, b, c, expected):  
    assert triangle_type(a, b, c) == expected
```

Parameters from a generator

```
def many_triangles():  
    for t in [ (90, 60, 30, "right"),  
              (100, 40, 40, "obtuse"),  
              (60, 60, 60, "acute"),  
              (0, 0, 0, "invalid")]:  
        yield t  
  
@pytest.mark.parametrize( 'a, b, c, expected', many_triangles())  
def test_func(a, b, c, expected):  
    assert triangle_type(a, b, c) == expected
```

Parameters from a file

triangle_data.csv

```
90,60,30,right  
100,40,40,obtuse  
60,60,60,acute  
0,0,0,invalid
```

```
import csv  
  
def many_triangles():  
    with open('triangle_data.csv') as csvfile:  
        for a, b, c, expected in csv.reader(csvfile):  
            yield (int(a), int(b), int(c), expected)  
  
@pytest.mark.parametrize( 'a, b, c, expected', many_triangles()  
def test_func(a, b, c, expected):  
    assert triangle_type(a, b, c) == expected
```

Back to a List

```
many_triangles = [  
    (90, 60, 30, "right"),  
    (100, 40, 40, "obtuse"),  
    (60, 60, 60, "acute"),  
    (0, 0, 0, "invalid")  
]  
  
@pytest.mark.parametrize('a, b, c, expected', many_triangles)  
def test_func(a, b, c, expected):  
    assert triangle_type(a, b, c) == expected
```

```
$ pytest test_7.py  
===== test session starts =====  
  
test_7.py::test_func[90-60-30-right] PASSED [ 25%]  
test_7.py::test_func[100-40-40-obtuse] PASSED [ 50%]  
test_7.py::test_func[60-60-60-acute] PASSED [ 75%]  
test_7.py::test_func[0-0-0-invalid] FAILED [100%]  
  
===== 1 failed, 3 passed in 0.03s =====
```


Run the last failing test case

```
$ pytest --tb=short --lf test_7.py
===== test session starts =====
collected 4 items / 3 deselected / 1 selected
run-last-failure: rerun previous 1 failure

test_7.py::test_func[0-0-0-invalid] FAILED [100%]

===== FAILURES =====
_____ test_func[0-0-0-invalid] _____
test_7.py:14: in test_func
    assert triangle_type(a, b, c) == expected
E   AssertionError: assert 'acute' == 'invalid'
E       - acute
E       + invalid
===== 1 failed, 3 deselected in 0.03s =====
```

Run test cases with 60 degree angles

```
$ pytest -k 60 test_7.py
===== test session starts =====

test_7.py::test_func[90-60-30-right] PASSED          [ 50%]
test_7.py::test_func[60-60-60-acute] PASSED          [100%]

===== 2 passed, 2 deselected in 0.01s =====
```

Run an individual test case

```
$ pytest test_7.py::test_func[0-0-0-invalid]
===== test session starts =====

test_7.py::test_func[0-0-0-invalid] FAILED          [100%]

===== 1 failed in 0.03s =====
```

Fixture Parametrization

Function: test_7.py

```
@pytest.mark.parametrize('a, b, c, expected', many_triangles)
def test_func(a, b, c, expected):
    assert triangle_type(a, b, c) == expected
```

Fixture: test_8.py

```
@pytest.fixture(params=many_triangles)
def a_triangle(request):
    return request.param

def test_fix(a_triangle):
    a, b, c, expected = a_triangle
    assert triangle_type(a, b, c) == expected
```

Object parameters get non-helpful names

```
many_triangles = [
    (90, 60, 30, "right"),
    (100, 40, 40, "obtuse"),
    (60, 60, 60, "acute"),
    (0, 0, 0, "invalid")]

@pytest.fixture(params=many_triangles)
def a_triangle(request):
    return request.param

def test_fix(a_triangle):
    a, b, c, expected = a_triangle
    assert triangle_type(a, b, c) == expected
```

```
$ pytest test_8.py
===== test session starts =====
test_8.py::test_fix[a_triangle0] PASSED [ 25%]
test_8.py::test_fix[a_triangle1] PASSED [ 50%]
test_8.py::test_fix[a_triangle2] PASSED [ 75%]
test_8.py::test_fix[a_triangle3] FAILED [100%]
===== 1 failed, 3 passed in 0.03s =====
```

An ids list

```
many_triangles = [  
    (90, 60, 30, "right"),  
    (100, 40, 40, "obtuse"),  
    (60, 60, 60, "acute"),  
    (0, 0, 0, "invalid") ]  
  
@pytest.fixture(params=many_triangles,  
                 ids=['right', 'obtuse', 'acute', 'invalid'])  
def a_triangle(request):  
    return request.param  
  
def test_fix(a_triangle):  
    a, b, c, expected = a_triangle  
    assert triangle_type(a, b, c) == expected
```

```
$ pytest test_9.py  
===== test session starts =====  
test_9.py::test_fix[right] PASSED [ 25%]  
test_9.py::test_fix[obtuse] PASSED [ 50%]  
test_9.py::test_fix[acute] PASSED [ 75%]  
test_9.py::test_fix[invalid] FAILED [100%]  
===== 1 failed, 3 passed in 0.03s =====
```

An ids function

```
many_triangles = [
    (90, 60, 30, "right"),
    (100, 40, 40, "obtuse"),
    (60, 60, 60, "acute"),
    (0, 0, 0, "invalid") ]

@pytest.fixture(params=many_triangles,
                ids=str) # or repr, or lambda t:t[3]
def a_triangle(request):
    return request.param

def test_fix(a_triangle):
    a, b, c, expected = a_triangle
    assert triangle_type(a, b, c) == expected
```

```
$ pytest test_10.py
===== test session starts =====
test_10.py::test_fix[(90, 60, 30, 'right')] PASSED [ 25%]
test_10.py::test_fix[(100, 40, 40, 'obtuse')] PASSED [ 50%]
test_10.py::test_fix[(60, 60, 60, 'acute')] PASSED [ 75%]
test_10.py::test_fix[(0, 0, 0, 'invalid')] FAILED [100%]
===== 1 failed, 3 passed in 0.03s =====
```

Custom ids function

```
def idfn(a_triangle):  
    a, b, c, expected = a_triangle  
    return f'{a}-{b}-{c}-{expected}'  
  
@pytest.fixture(params=many_triangles, ids=idfn)  
def a_triangle(request):  
    return request.param  
  
def test_fix(a_triangle):  
    a, b, c, expected = a_triangle  
    assert triangle_type(a, b, c) == expected
```

```
$ pytest test_11.py  
===== test session starts =====  
test_11.py::test_fix[90-60-30-right] PASSED [ 25%]  
test_11.py::test_fix[100-40-40-obtuse] PASSED [ 50%]  
test_11.py::test_fix[60-60-60-acute] PASSED [ 75%]  
test_11.py::test_fix[0-0-0-invalid] FAILED [100%]  
===== 1 failed, 3 passed in 0.03s =====
```


pytest_generate_tests()

```
def pytest_generate_tests(metafunc):  
    if "gen_triangle" in metafunc.fixturenames:  
        metafunc.parametrize("gen_triangle",  
                               many_triangles,  
                               ids=idfn)
```

```
def test_gen(gen_triangle):  
    a, b, c, expected = gen_triangle  
    assert triangle_type(a, b, c) == expected
```

```
$ pytest test_12.py  
===== test session starts =====  
  
test_12.py::test_gen[90-60-30-right] PASSED [ 25%]  
test_12.py::test_gen[100-40-40-obtuse] PASSED [ 50%]  
test_12.py::test_gen[60-60-60-acute] PASSED [ 75%]  
test_12.py::test_gen[0-0-0-invalid] FAILED [100%]  
  
===== 1 failed, 3 passed in 0.03s =====
```

metafunc

From docs.pytest.org/en/latest/reference.html#metafunc

- Metafunc objects are passed to the `pytest_generate_tests` hook.
- They help to inspect a test function and to generate tests according to
 - test configuration
 - or values specified in the class or module where a test function is defined.

test.param

```
smoke = pytest.mark.smoke

many_triangles = [
    pytest.param(90, 60, 30, "right", marks=smoke),
    pytest.param(100, 40, 40, "obtuse", marks=smoke),
    (90, 60, 30, "right"),
    pytest.param(0, 0, 0, "invalid", id='zeros'),
]
```

```
$ pytest -m smoke test_13.py
===== test session starts =====
test_13.py::test_func[90-60-30-right] PASSED [ 50%]
test_13.py::test_func[100-40-40-obtuse] PASSED [100%]
===== 2 passed, 4 deselected in 0.01s =====
```

```
$ pytest -k zeros test_13.py
===== test session starts =====
test_13.py::test_func[zeros] PASSED [100%]
===== 1 passed, 3 deselected in 0.01s =====
```

indirect parameter

```
@pytest.fixture()
def expected(request):
    if request.param == 'obtuse':
        print('\nthis is one of the obtuse cases')
    return request.param

@pytest.mark.parametrize('a, b, c, expected', many_triangles,
                          indirect=['expected'])
def test_func(a, b, c, expected):
    assert triangle_type(a, b, c) == expected
```

The parameter value goes through a fixture before making it to the test, an "indirect" route.

indirect example

```
def many_triangles():
    with open('triangle_data.csv') as csvfile:
        for a, b, c, expected in csv.reader(csvfile):
            yield (a, b, c, expected)

@pytest.fixture()
def a(request):
    return int(request.param)

@pytest.fixture()
def b(request):
    return int(request.param)

@pytest.fixture()
def c(request):
    return int(request.param)

@pytest.mark.parametrize('a, b, c, expected', many_triangles(),
                          indirect=['a', 'b', 'c'])
def test_func(a, b, c, expected):
    assert triangle_type(a, b, c) == expected
```

More test cases

```
many_triangles = [  
    ( 1, 1, 178, "obtuse"), # big angles  
    ( 91, 44, 45, "obtuse"), # just over 90  
    (0.01, 0.01, 179.98, "obtuse"), # floats  
  
    (90, 60, 30, "right"), # check 90 for each angle  
    (10, 90, 80, "right"),  
    (85, 5, 90, "right"),  
  
    (89, 89, 2, "acute"), # just under 90  
    (60, 60, 60, "acute"),  
  
    (0, 0, 0, "invalid"), # zeros  
    (61, 60, 60, "invalid"), # sum > 180  
    (90, 91, -1, "invalid"), # negative numbers  
]
```

For more on test case selection:

- [Test & Code 38](#) : Prioritize software tests with RCRCRC
- [Test & Code 39](#) : equivalence partitioning, boundary value analysis, decision tables

Review

```
@pytest.mark.parametrize('a, b, c, expected', many_triangles)
def test_func(a, b, c, expected):
    assert triangle_type(a, b, c) == expected
```

```
@pytest.fixture(params=many_triangles, ids=idfn)
def a_triangle(request):
    return request.param

def test_fix(a_triangle):
    a, b, c, expected = a_triangle
    assert triangle_type(a, b, c) == expected
```

```
def pytest_generate_tests(metafunc):
    if "gen_triangle" in metafunc.fixturenames:
        metafunc.parametrize("gen_triangle",
                               many_triangles, ids=idfn)

def test_gen(gen_triangle):
    a, b, c, expected = gen_triangle
    assert triangle_type(a, b, c) == expected
```

Combining Techniques

You can have multiple parametrizations for a test function.

- can have multiple `@pytest.mark.parametrize()` decorators.
- can parameterize multiple fixtures per test
- can use `pytest_generate_tests()` to parametrize multiple parameters
- can use a combination of techniques
- can blow up into lots and lots of test cases very fast

Resources

- [Python Testing with pytest](#)
 - The fastest way to get super productive with pytest
- [pytest docs](#) on
 - [parametrization](#), in general
 - [function parametrization](#)
 - [fixture parametrization](#)
 - [pytest_generate_tests](#)
 - [indirect](#)
- [podcasts](#)
 - [Test & Code](#)
 - [Python Bytes](#)
 - [Talk Python](#)
- [slack community](#): [Test & Code Slack](#)
- [Twitter](#): [@brianokken](#),
- [Training](#): testandcode.com/training
- This code, and markdown for slides, on github.com/okken/talks

