About Ranks

TAU 100

Learning Paths

TAU Slack M

Certificates

Karen Bocardo

Logout

```
Sometimes, the behaviors we want to test can have different kinds of inputs and outputs. They may also have specific boundary cases that should be tested.
Let's explore this by writing new tests for multiplication. And let's think before we code.
I'm going to write different multiplication test case ideas and comments in "test_math.py".
```

test_math.py — tau-intro-to-pytest

Multiplication test ideas

two positive integers

• We could multiply two positive integers.

• We could test identity by multiplying any number by one.

test_math.py

43

44

45

tests > 📌 test_math.py > ...

• We could test the zero property by multiplying any number by zero.

• We can multiply a positive by a negative.

Transcripted Summary

• We could test negative numbers multiplied by negative numbers.

✓ OPEN EDITO... 1 UNSAVED

test_math.p... M

∨ TAU-INTRO-TO-PYTEST

• We could also multiply floating point numbers instead of integers.

C? **EXPLORER**

√ tests

gg. # identity: multiplying any number by 1 46 test_math.py M # zero: multiplying any number by 0 47 ≡ .coverage # positive by a negative .gitignore # negative by a negative LICENSE # multiply floats README.md (8) > OUTLINE TIMELINE > MAVEN PROJECTS Python 3.8.1 64-bit ⊗ 0 △ 0 Ln 50, Col 18 Spaces: 2 UTF-8 LF Python 🖓 🚨 These are what we call "equivalence classes" of test case inputs. Each one represents a unique kind of input that yields a unique kind of outcome. A good test suite provides one test case for each equivalence class of inputs for a behavior under test. For example, the "identity" equivalence class could be represented by testing the inputs 1 and 99 (1 x 99). Adding an additional test for inputs 1 and 100 (1 x 100) could be considered repetitive because the equivalence class for "identity" is already covered by the first test.

In pytest, we could add these six tests as six separate test functions:

def test_multiply_two_positive_ints():

assert 2 * 3 == 6

def test_multiply_zero():

EXPLORER

√ tests

✓ OPEN EDITO... 1 UNSAVED

test_math.p... M

∨ TAU-INTRO-TO-PYTEST

test_math.py

assert 0 * 100 == 0

C

parametrized inputs.

Unnecessary tests should be avoided because they add time and cost for little value in return.

def test_multiply_identity(): assert 1 * 99 == 99

Multiplication test ideas

identity: multiplying any number by 1

two positive integers

ξ'n

0 • • test_math.py — tau-intro-to-pytest

test_math.py

43

44

45

46

tests > dest_math.py > ...

```
# zero: multiplying any number by 0
            # positive by a negative
            .gitignore
                                               # negative by a negative
                                         49
            1 LICENSE
                                               # multiply floats
                                         50
                                         51
           ① README.md
   品
                                         52
                                               def test_multiply_two_positive_ints():
                                         53
                                                  assert 2 * 3 == 6
                                         54
                                         55
                                               def test_multiply_identity():
                                         56
                                                 assert 1 * 99 == 99
                                         57
                                         58
                                               def test_multiply_zero():
   (8)
                                         59
                                                 assert 0 * 100 == 0
                                         60
          > OUTLINE
            TIMELINE
            MAVEN PROJECTS
                                   Python 3.8.1 64-bit
                                                      ⊗ 0 △ 0
                                                                                                 Spaces: 2 UTF-8 LF Python 尽 🚨
                                                                                     Ln 60, Col 3

    example/4-parametrize*

However, after writing the first few tests, we can see how repetitive the code becomes.
These tests violate the DRY principle - "Don't Repeat Yourself" ...
Notice how function names and test calls are very similar.
Since these examples are very simple, the problem may not seem too bad, but in the real world, tests typically have several lines of code.
Code duplication becomes code cancer when programmers copy and paste code with bad practices.
Maintaining multiple copies of the same logic becomes burdensome, too.
Thankfully, pytest has a better pattern: "@pytest.mark.parametrize". Using "pytest.mark.parametrize", we can write one test function that takes multiple
```

Parametrizing fixtures and test functions

· @pytest.mark.parametrize allows one to define multiple sets of arguments and fixtures at the test

The builtin pytest.mark.parametrize decorator enables parametrization of arguments for a test function.

Here is a typical example of a test function that implements checking that a certain input leads to an ex-

@pytest.mark.parametrize("test_input,expected", [("3+5", 8), ("2+4", 6), ("6*9", 42)])

pytest enables test parametrization at several levels: · pytest.fixture() allows one to parametrize fixture functions.

Contents

Examples Customize

Changelog

Contributing

Sponsor

Backwards Compatibility

pytest for Enterprise

Python 2.7 and 3.4 Support

· pytest_generate_tests allows one to define custom parametrization schemes or extensions. Go Search **Table Of Contents** @pytest.mark.parametrize: parametrizing test functions Home

content of test_expectation.py

def test_eval(test_input, expected):

assert eval(test_input) == expected

function or class.

pected output:

import pytest

```
License
                                              Here, the @parametrize decorator defines three different (test_input, expected) tuples so that the
                     Contact Channels
                                              test_eval function will run three times using them in turn:
                     Parametrizing fixtures and test
                     functions
                                               $ pytest
                                                @pytest.mark.parametrize:
                                                platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
                       parametrizing test functions
                                                cachedir: $PYTHON_PREFIX/.pytest_cache

    Basic

                                                rootdir: $REGENDOC_TMPDIR
                       pytest_generate_tests
                                                collected 3 items
                       example

    More examples

                                                                                                          [100%]
                                                test_expectation.py ..F
                     Index
                                                @pytest.mark.parametrize
Let's rewrite our multiplication tests using "@pytest.mark.parametrize". Make sure that your module already imports pytest.
First, let's write a list of tuples in which each tuple represents an equivalent class of inputs and outputs. Write one tuple for each equivalence class.
  products = [
    (2, 3, 6),
                                         # postive integers
    (1, 99, 99),
                                         # identity
    (0, 99, 0),
                                         # zero
    (3, -4, -12),
                                        # positive by negative
    (-5, -5, 25),
                                        # negative by negative
    (2.5, 6.7, 16.75)
                                         # floats
Second, let's write one test case function for multiplication named "test_multiplication".
Unlike other test functions, this one needs arguments.
Let's name them a and b for the inputs and product for the outputs. Inside this test function, let's add the line assert a * b == product to test multiplication.
```

Next, we need to make pytest pass the parametrized values into the new test case. That's where we use @pytest.mark.parametrize. Make sure import pytest is already in the test module. @pytest.mark.parametrize is a decorator for the test multiplication function. In Python, a decorator is a special function that wraps around another function. It is a simple form of aspect-oriented programming ...

@pytest.mark.parametrize('a, b, product', products)

These names must match the parameter names for the test case function, a, b, and product.

And just like that, we have parametrized one test case function to cover multiple sets of inputs.

sterling2:tau-intro-to-pytest andylpk247\$

When pytest runs tests, it will run this test function six times, once for each tuple in the parameter list.

Even though our test module has only four tests functions, pytest runs a total of nine tests. Awesome.

For example, in the first tuple (2, 3, 6), a will be 2, b will be 3, and product will be 6. Let's run our new tests.

def test_multiplication(a, b, product):

def test_multiplication(a, b, product):

"parameterize". Be careful not to make a typo.

assert a * b == product

assert a * b == product

Don't worry if you don't fully understand how decorators work, just know that we can use @pytest.mark.parametrize to run a test case with multiple inputs. **NOTE**

I'd like to make a brief comment about spelling. The decorator uses the British English spelling, "parametrize", not the American English spelling,

We also need to pass two arguments to the decorator. The first argument is a string containing a comma-separated list of variable names.

For @pytest.mark.parametrize, the inner function is the test case. The outer function is the decorator itself, and it will call the inner test case once per input tuple.

tau-intro-to-pytest — -bash — 101×28 sterling2:tau-intro-to-pytest andylpk247\$ python -m pytest platform darwin -- Python 3.8.1, pytest-5.4.3, py-1.8.1, pluggy-0.13.1 rootdir: /Users/andylpk247/Programming/automation-panda/tau-intro-to-pytest plugins: metadata-1.9.0, bdd-3.4.0, cov-2.10.0, html-2.1.1, forked-1.1.3, xdist-1.32.0 collected 9 items tests/test_math.py [100%]

The second argument is the list of parametrized values. Note that the number of variable names and the length of each tuple in the list is three. These must match.

```
Pytest parameters make it easy to do data-driven testing in which the same test cases crank through multiple inputs.
  This test case is just a basic example of what you can do with parameters.
  You can use any Python object type for parameter values, not just integers.
  Since parameters are passed into the test cases as a list, you could also store data in external files like CSVs or Excel spreadsheets and read them in when the test
  runs.
  There are a bunch of other advanced tricks you can do with parameters. I won't cover them in this course, but you can look them up online in the pytest docs.
  If you want to take test parametrization a step further, look into Property-Based Testing with Hypothesis ...
  Hypothesis is a testing library that can integrate with pytest. With Hypothesis, you can specify properties of parameter values rather than hard coding parameter
  values yourself.
  When tests run, Hypothesis then cranks through several matching values, up to hundreds or thousands of generated tests.
  Property-Based Testing isn't the best approach for all types of testing but it certainly is worth learning.
  Note that we will not cover Hypothesis further in this course.
  Resources
pytest parameters 
More pytest parameters 

    Don't Repeat Yourself ☐

Python decorators

    Aspect-Oriented Programming 

Hypothesis ☑
```

Hide my answers

