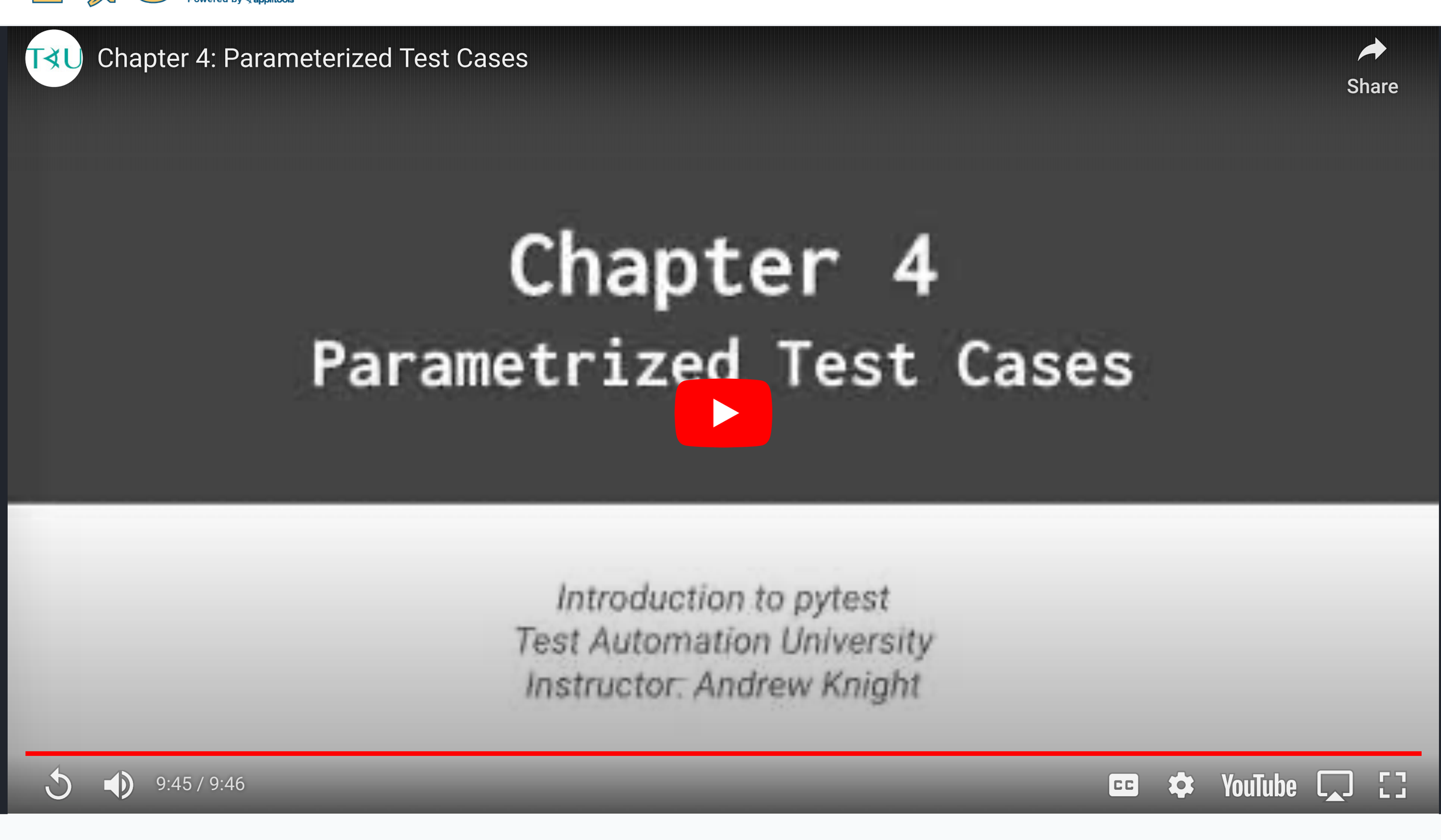


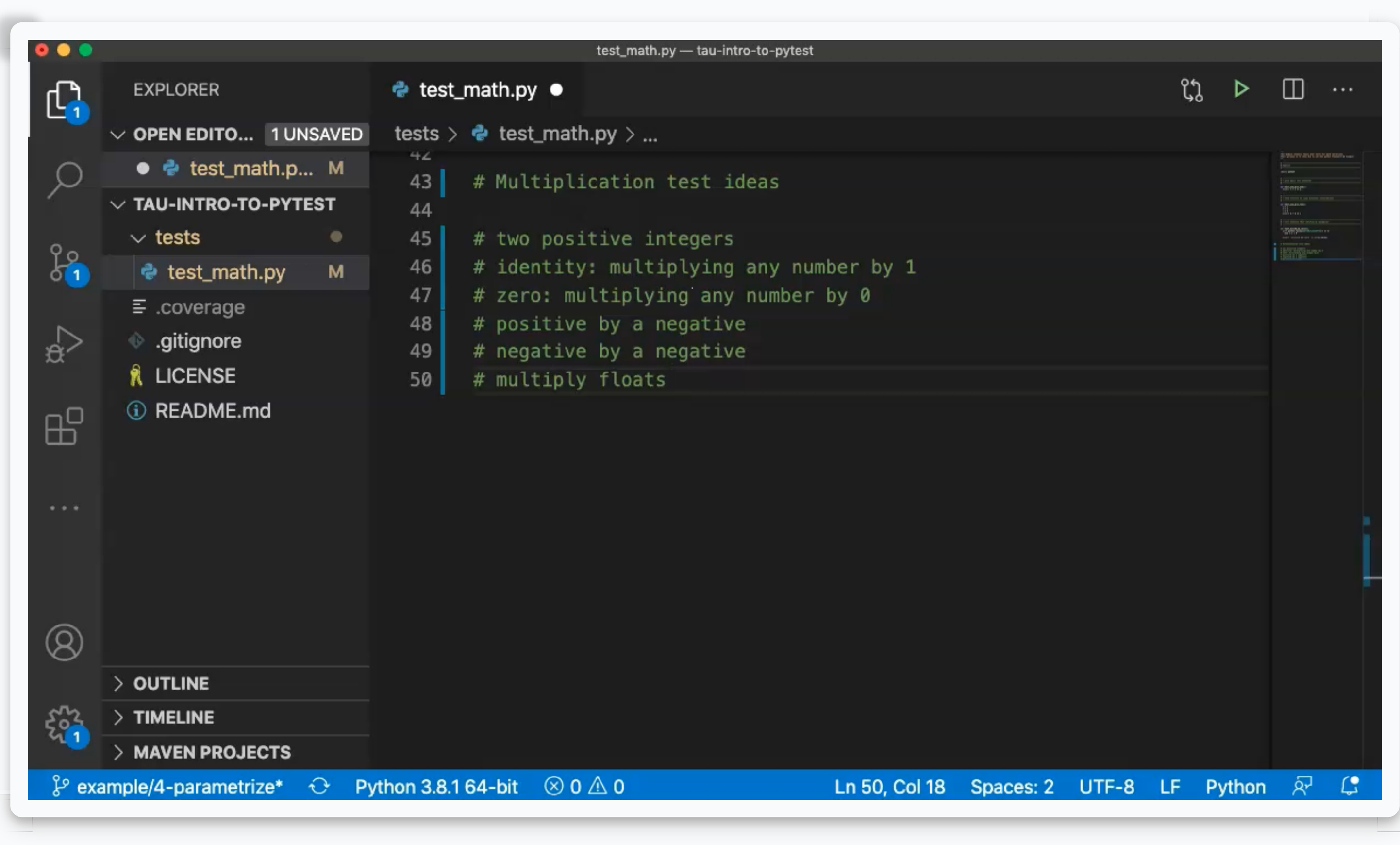
▼ Transcripted Summary

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".

- We could multiply two positive integers.
- We could test identity by multiplying any number by one.
- We could test the zero property by multiplying any number by zero.
- We can multiply a positive by a negative.
- We could test negative numbers multiplied by negative numbers.
- We could also multiply floating point numbers instead of integers.



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.

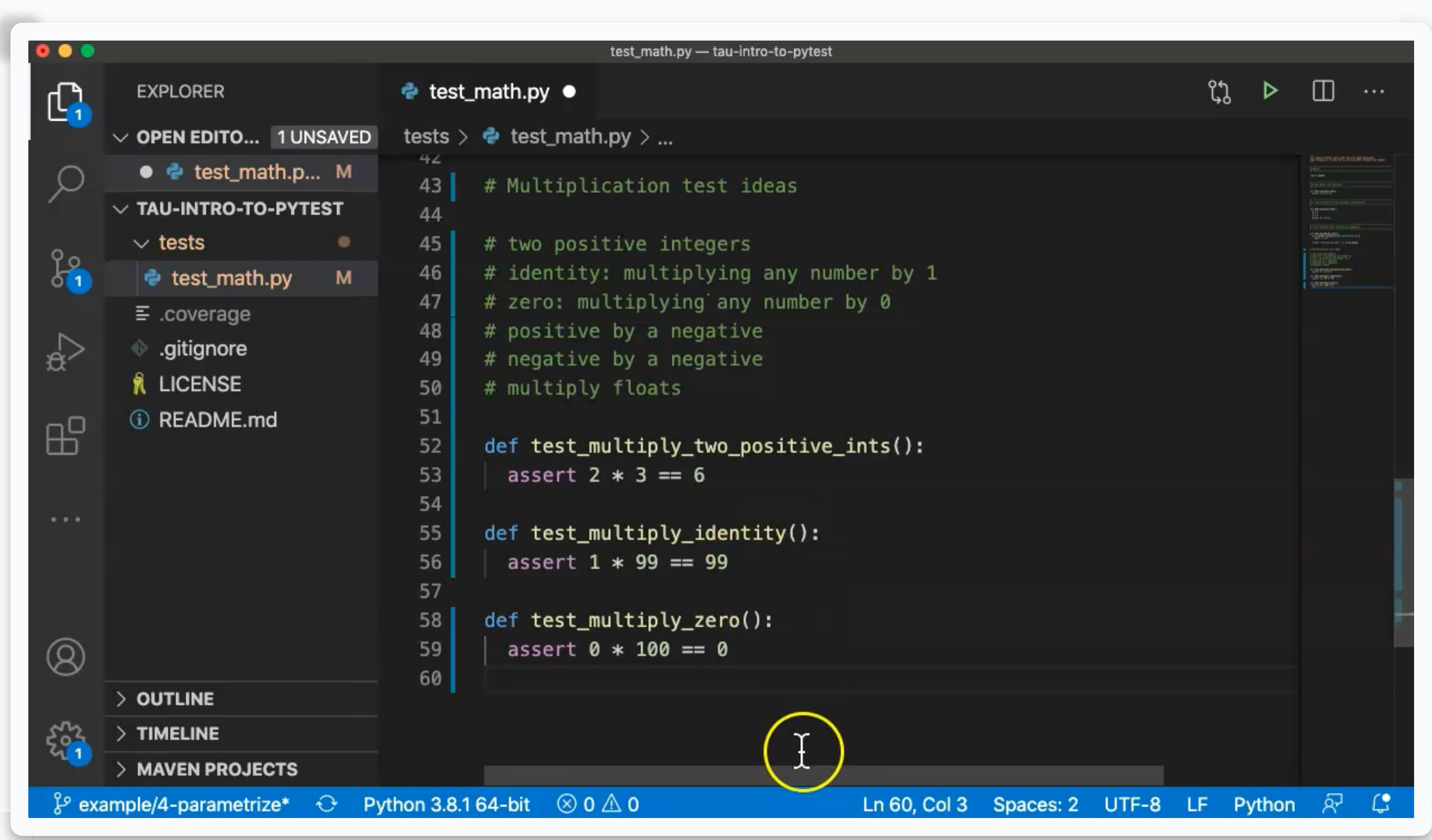
Unnecessary tests should be avoided because they add time and cost for little value in return.

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

```py
def test_multiply_two_positive_ints():
  assert 2 * 3 == 6

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

def test_multiply_zero():
  assert 0 * 100 == 0
```



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".

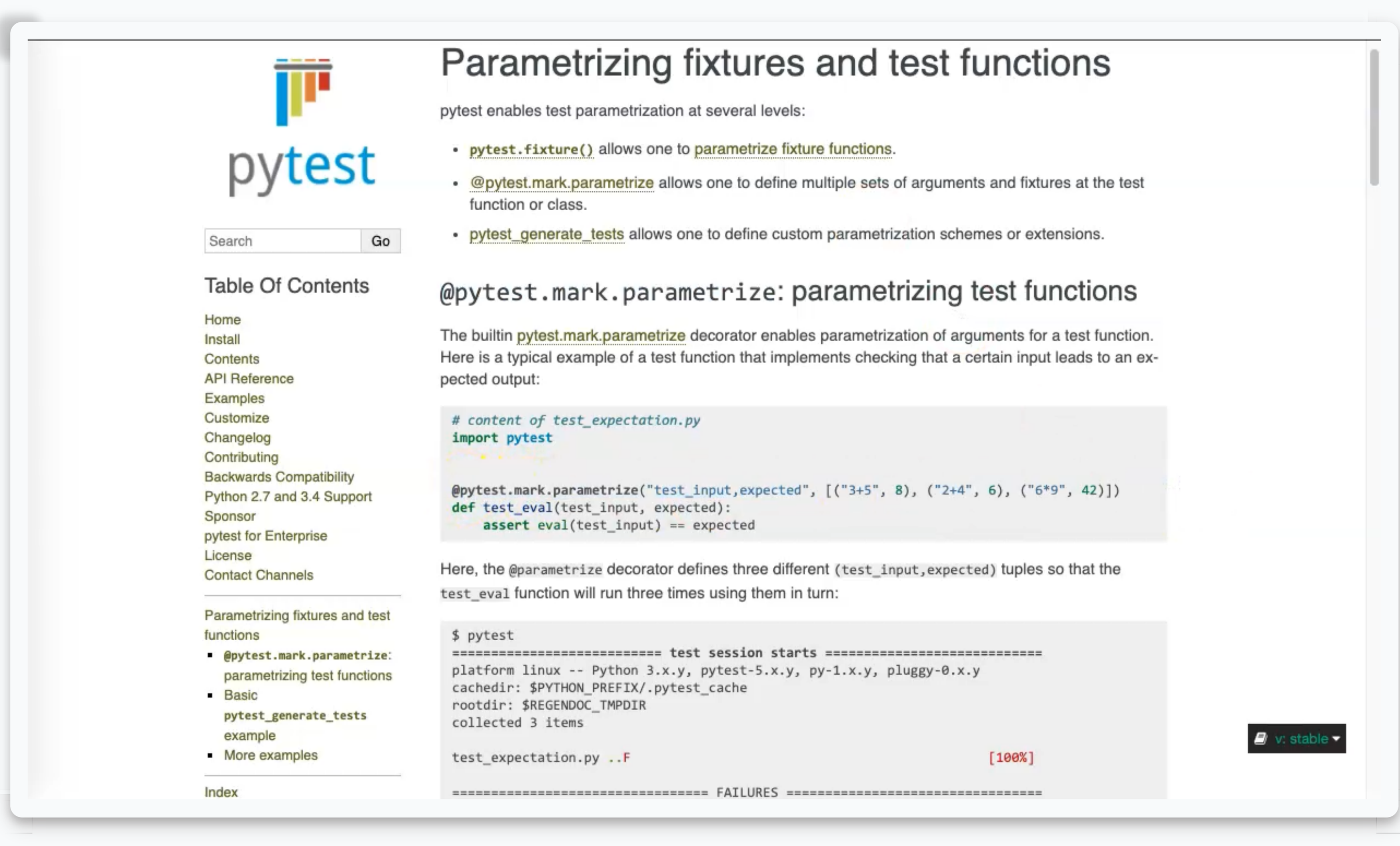
Notice how function names and test calls are very similar.

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 parametrized inputs.



@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.

```py
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.

```py
def test_multiplication(a, b, product):
  assert a * b == product
```

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.

```py
@pytest.mark.parametrize('a, b, product', products)
def test_multiplication(a, b, product):
  assert a * b == product
```

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.

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, "paramet**e**rize". Be careful not to make a typo.

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

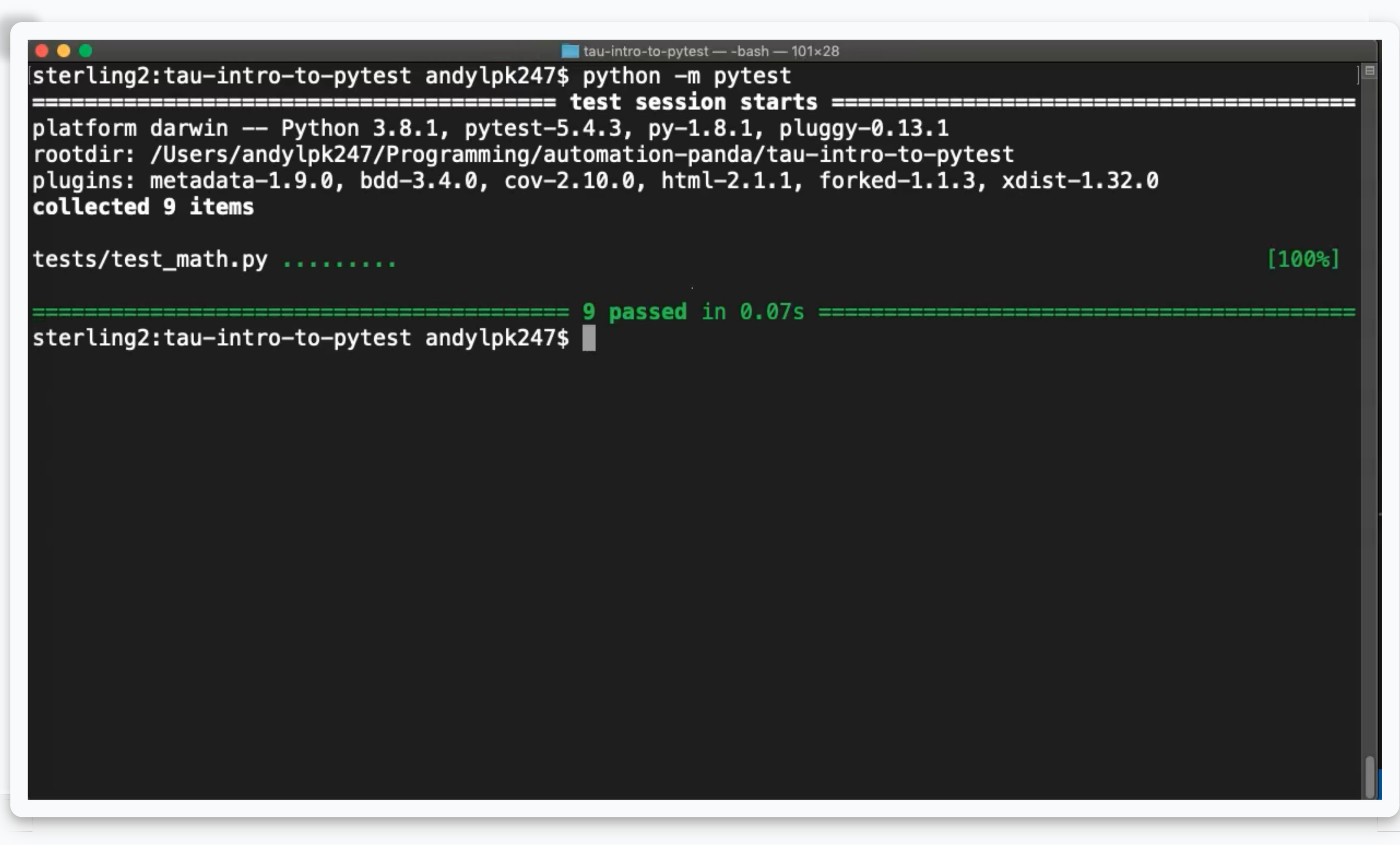
These names must match the parameter names for the test case function, `a`, `b`, and `product`.

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.

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

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

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.



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

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

1. What is an "equivalence class" for test case inputs?

- ○ A range of possible output values.
- ○ Duplicate test parameters.
- ○ A hard-coded set of input values.
- ● A unique kind of input that yields a unique kind of output.

2. What is the name of the pytest decorator for parametrized inputs?

- ○ @pytest.parametrize
- ● @pytest.mark.parametrize
- ○ @pytest.mark.parameterize
- ○ @pytest.parameters

3. What is the data format for passing pytest parameters?

- ○ a list of Parameter objects
- ○ a one-dimensional list of parameter values
- ○ a dictionary of parameter names to values
- ● a list of parameter value tuples

4. pytest parameters may be strings as well as numbers.

- ● true
- ○ false

5. pytest parameters cannot be read from external files.

- ○ true
- ● false

Note: 100 credits is for successful completion on the first try; 50 credits for the second try, and 25 credits thereafter

Prev Chapter | Next Chapter

Share | Share