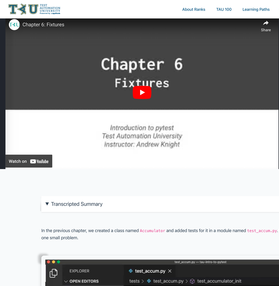
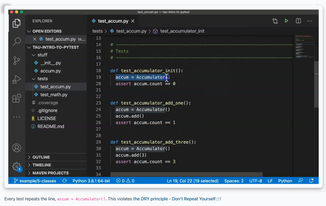
Chapter 6: Fixtures

Share



▼ Transcripted Summary

In the previous chapter, we created a class named `Accumulator` and added tests for it in a module named `test_accum.py`. If we review our test code, we will notice one small problem.



Every test repeats the line, `accum = Accumulator()`. This violates the DRY principle = Don't Repeat Yourself.

Automated test cases frequently repeat steps because many tests share the same operations. For example, every test here needs an accumulator object.

Whenever we find ourselves repeating code, we should try to find a better way to implement it.

Thankfully, pytest provides a nifty solution for test setup: fixtures!

Fixtures are special functions that pytest can call before test case functions.

They're the best way to handle "Arrange" steps shared by multiple tests in the context of the "Arrange-Act-Assert" pattern.

Let's refactor our Accumulator tests to use a fixture that creates an Accumulator object.

Fixtures are functions.

Between the import statements and the test case functions, add a new function named `accum`.

In its body, add one line: `return Accumulator()`. Decorate it with the `@pytest.fixture` decorator so that pytest knows it's a fixture function.

```
@pytest.fixture
def accum():
  return Accumulator()
```

This `accum` fixture is concise because the only thing it needs to do is create a new Accumulator object.

Importantly, note that the function _returns_ the newly constructed object. It does not assign the object to a global variable. A fixture should always return a value.

Now that we have a fixture, let's update the test cases.

Look at `test_accumulator_init()`.

Remove the object creation line `accum = Accumulator()` and add a parameter to the test function signature named `accum`.

This is all we need to do to make this test case use this fixture.

```
def test_accumulator_init(accum):
  assert accum.count == 0
```

How does the fixture work? It's pytest magic!

When pytest discovers a test case function, it looks at the function's parameter list.

If the function has parameters, then it will search for fixtures to match each parameter's name.

In our case, the test function has a parameter named `accum`, so pytest looks for a fixture named `accum` which it will find in the same module.

Pytest will then execute the fixture and pass the fixture's return value into the test case function.

Thus, in our test case, the `accum` variable will refer to the new `Accumulator` object created by the `accum` fixture. Nifty.

This is a clever form of _dependency injection_.

The test case doesn't set up or "arrange" the test objects itself.

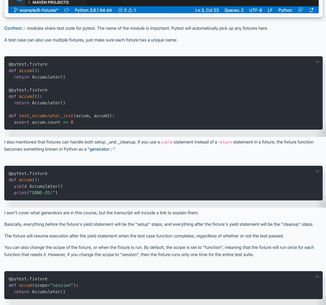
Instead, the fixture handles setup and injects the required objects as dependencies into the test function.

This separation of concerns makes test cases more readable, more consistent, and more maintainable.

It also makes new test cases easier to write. Let's update the remainder of the tests using the "accum" fixture.

```
def test_accumulator_init(accum):
  assert accum.count == 0

def test_accumulator_add_one(accum):
  accum.add()
  assert accum.count == 1

def test_accumulator_add_three(accum):
  accum.add()
  assert accum.count == 3

def test_accumulator_add_twice(accum):
  accum.add()
  accum.add()
  assert accum.count == 2

def test_accumulator_cannot_set_count_directly(accum):
  with pytest.raises(AttributeError, match="can't set attribute") as e:
    accum.count = 10
```

Re-run the tests to make sure they still work.



Everything passes. Great!

pytest versus xUnit Frameworks

Fixtures may seem confusing at first. Out of every feature, fixtures make pytest unique amidst other test frameworks that are part of the xUnit family, like Python's unittest, Java's JUnit and C#'s NUnit. xUnit frameworks all follow similar conventions.

Tests are written as classes instead of functions.

A test class has methods for individual test cases.

They also have setup and cleanup methods.

When tests run, setup and cleanup methods are executed before and after each test case method individually.

xUnit-style test classes provide a decent structure for automating tests but, in my opinion, they have inherent weaknesses.

A test class's setup and cleanup methods can be used only within that class.

They cannot be reused by other classes. Classes and their variables also require programmers to carefully manage state in between test phases.

For example, if a particular variable doesn't get initialized, then the automation could crash and yield an unintuitive failure message.

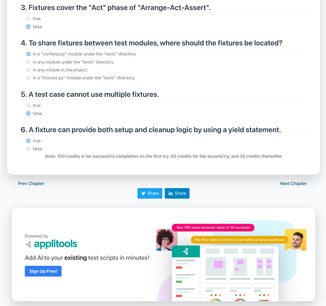
pytest avoids the limitations of classes by structuring tests as functions.

Fixtures are simply the function-based way to handle setup and cleanup operations.

Fixtures can be used by any test function in any module, so they are universally shareable since they use dependency injection to share state, they protect tests against unintended side effects.

Advanced Fixture Features

There are a few advanced tricks you can do with fixtures as well. If you want to share fixtures between multiple test modules, you can move it to a module in the "tests" directory named "conftest.py".



Conftest modules share test code for pytest. The name of the module is important. Pytest will automatically pick up any fixtures here.

A test case can also use multiple fixtures, just make sure each fixture has a unique name:

```
@pytest.fixture
def accum1():
  return Accumulator()

@pytest.fixture
def accum2():
  return Accumulator()

def test_accumulator_init(accum, accum2):
  assert accum.count == 0
```

I also mentioned that fixtures can handle both setup _and_ cleanup. If you use a `yield` statement instead of a `return` statement in a fixture, the fixture function becomes something known in Python as a "generator".

```
@pytest.fixture
def accum():
  yield Accumulator()
  print("DONE-ZO!")
```

I won't cover what generators are in this course, but the transcript will include a link to explain them.

Basically, everything before the fixture's yield statement will be the "setup" steps, and everything after the fixture's yield statement will be the "cleanup" steps.

The fixture will resume execution after the yield statement when the test case function completes, regardless of whether or not the test passed.

You can also change the scope of the fixture, or when the fixture is run. By default, the scope is set to "function", meaning that the fixture will run once for each function that needs it. However, if you change the scope to "session", then the fixture runs only one time for the entire test suite.

```
@pytest.fixture
def accum(scope="session"):
  return Accumulator()
```

If multiple tests use the fixture, then the fixture will run only for the first test. pytest will then store its return value and simply inject the return value into each subsequent test that needs it.

"Session" scope would not be appropriate for these Accumulator tests, but it would be appropriate for a fixture that needs to read data from an external file. Other scope levels include "class", "module", and "package".

Finally, pytest provides several fixtures out of the box.

- `monkeypatch` can be used for modifying classes, functions, and other objects
- `request` provides test case metadata
- `tmpdir` and `tmp_path` provide temporary directories

pytest plugins may also provide additional fixtures.

Whether you write your own fixtures or use existing ones, fixtures are an indispensable part of the pytest framework.

Resources

- pytest fixtures
- DRY Principle
- Dependency Injection
- conftest.py
- Python generators

Hide my answers

1. In pytest, what is a fixture?

- Any function in a test module that is not a test case.
- A function that handles setup and cleanup operations for a test case.
- A term that refers to the overall structure of a test case module.
- Another name for a test case function.

2. What is the name of the decorator used for defining functions as fixtures?

- @pytest.fixture
- @pytest.mark.fixture
- @pytest.fix
- @fixture

3. Fixtures cover the "Act" phase of "Arrange-Act-Assert".

- true
- false

4. To share fixtures between test modules, where should the fixtures be located?

- In a "conftest.py" module under the "tests" directory.
- In any module under the "tests" directory.
- In any module in the project.
- In a "fixtures.py" module under the "tests" directory.

5. A test case cannot use multiple fixtures.

- true
- false

6. A fixture can provide both setup and cleanup logic by using a yield statement.

- true
- false

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