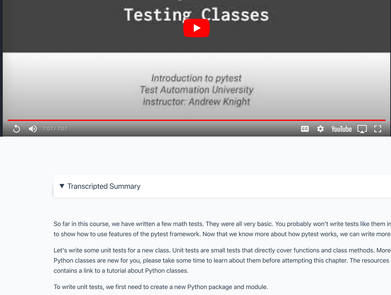
Chapter 5: Testing Classes

Share



▼ Transcripted Summary

So far in this course, we have written a few math tests. They were all very basic. You probably won't write tests like them in the real world. However, we wrote them to show how to use features of the pytest framework. Now that we know more about how pytest works, we can write more realistic, meaningful test cases.

Let's write some unit tests for a new class. Unit tests are small tests that directly cover functions and class methods. More generally, they cover units of work. If Python classes are new for you, please take some time to learn about them before attempting this chapter. The resources section at the end of the transcript contains a link to a tutorial about Python classes.

To write unit tests, we first need to create a new Python package and module.

From the project root directory, create a new directory named "stuff".

Inside the "stuff" directory, create a new file named `__init__.py`.

In Python, any directory with a file named `__init__.py` is treated as a package, and any modules inside that package may be imported by other modules. Leave this file blank.

At this point, you may be wondering why our "tests" directory does not have a file named `__init__.py`. pytest does not require tests to be a package. In fact, making the "tests" directory a package may have unintended consequences with tools like tox.

> **NOTE**
>
> In Python, "dunder" is a colloquialism referring to double underscores.

Inside the `stuff` package, create another new file named `accum.py`. Inside this new module, we will add the code for a new class named `Accumulator`.

```
class Accumulator:

  def __init__(self):
    self._count = 0

  @property
  def count(self):
    return self._count

  def add(self, more=1):
    self._count += more
```

The Accumulator class is very simple. It saves a tally of numbers.

The `__init__` method initializes the class with a starting count of zero.

Internally, the tally is saved in the `self._count` variable. This variable should be treated as private because it is prefixed with a single underscore.

The `count` method returns the value of the count. This method is a property, as denoted by the `@property` decorator.

In Python, properties control how callers can "get" and "set" values. With this property, a caller can get the value of `count` but cannot set the value directly with an assignment statement.

Finally, the `add` method is the only way to change the internal count value. It accepts an amount to add as input and adds this amount to the internal account. By default, the amount to add is one, but this value may be overwritten.

Now that we have a class, let's write some unit tests for it. Create a new module named `test_accum.py` under the `tests` directory. In this module, add import statements for pytest and for the new Accumulator class:

```
import pytest
from stuff.accum import Accumulator
```

Then add five new test functions:

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


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


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


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


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

- Method `test_accumulator_init()` verifies that the new instance of the Accumulator class has a starting count of zero.
- Method `test_accumulator_add_one()` verifies that the `add()` method adds one to the internal count when it is called with no other arguments.
- Method test_accumulator_add_three() verifies that the `add()` method adds 3 to the count when it is called with the argument of 3.
- Method `test_accumulator_add_twice()` verifies that the count increases appropriately with multiple `add()` calls.
- Finally, method `test_accumulator_cannot_set_count_directly()` verifies that the `count` attribute cannot be assigned directly because it is a read-only property. Notice how we use `pytest.raises` to verify the attribute error.

Take a moment to review and study these tests functions.

You will notice that all of these unit tests follow a common pattern.

They construct an `Accumulator` object, they make calls to the `Accumulator` object, and they verify the counts of the `Accumulator` objects or else verify some error.

This pattern is called "Arrange-Act-Assert". It is the classic three-step pattern for functional test cases.

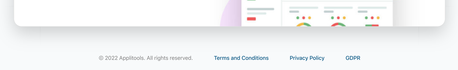
1. *Arrange* assets for the test (like a setup procedure).
2. *Act* by exercising the target behavior.
3. *Assert* that expected outcomes happened.

Remember this pattern whenever you write test cases. Following this pattern will keep your tests simple, focused, and valuable. It will also help you separate tests by unique behaviors.

Notice how none of our tests take any more `Act` steps after their `Assert` steps.

Separate, small, independent tests make failure analysis easier in the event of a regression.

Let's run our new accumulator unit tests.



This time, when we run pytest, we will see an additional line of dots for `tests/test_accum.py`. Despite now having 14 total tests, execution time is still sub-second. Very nice.

Resources

- Python classes
- Good pytest integration practices
- Python property decorator
- tox
- Arrange-Act-Assert

Hide my answers

1. What are "unit tests"?

- Small tests that cover variable assignments
- Large tests that cover Web UIs and REST APIs
- **Small tests that directly cover "units of work" like functions and methods**
- Manual tests that interact with the product under test like a user

2. What file turns a regular directory into a package in a Python project?

- _init_.py
- init.py
- **__init__.py**
- __init.py

3. pytest can run tests from multiple modules in the same Python project.

- **true**
- false

4. Which of the following lines represents an "Arrange" step?

- accum.add()
- accum.add(3)
- **accum = Accumulator()**
- assert accum.count == 0

5. Which of the following lines represents an "Assert" step?

- accum.add()
- accum = Accumulator()
- **assert accum.count == 0**
- accum.add(3)

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