Extracted from:

Python Testing with pytest, Second Edition

Simple, Rapid, Effective, and Scalable

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Python Testing with pytest

Second Edition

Simple, Rapid, Effective, and Scalable

Brian Okken

edited by Katharine Dvorak

Python Testing with pytest, Second Edition

Simple, Rapid, Effective, and Scalable

Brian Okken

The Pragmatic Bookshelf

Raleigh, North Carolina

ISBN-13: 978-1-68050-860-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—June 18, 2021

CHAPTER 10

Mocking

In the last chapter, we tested the Cards project through the API. In this chapter, we're going to test the CLI. When we wrote the test strategy for the Cards project in Writing a Test Strategy, on page ?, we included the following statement:

- Test the CLI enough to verify the API is getting properly called for all features.

We're going to use the mock package to help us with that. Shipped as part of the Python standard library as unittest.mock as of Python 3.3,[1] the mock package is used to swap out pieces of the system to isolate bits of our code under test from the rest of the system. *Mock objects* are sometimes called *test doubles*, *spies*, *fakes*, or *stubs*. Between pytest's own monkeypatch fixture (covered in Using monkeypatch, on page ?) and mock, you should have all the test double functionality you need.

In this chapter, we'll take a look at using mock to help us test the Cards CLI. We'll also look at using the CliRunner provided by Typer to assist in testing.

Isolating the Command-Line Interface

The Cards CLI uses the Typer library[2] to handle all of the command-line parts, and then it passes the real logic off to the Cards API. In testing the Cards CLI, the idea is that we'd like to test the code within cli.py and cut off access to the rest of the system. To do that, we have to look at cli.py to see how it's accessing the rest of Cards.

1. https://docs.python.org/3/library/unittest.mock.html
2. https://pypi.org/project/typer

The cli.py module accesses the rest of the Cards system through an import of cards:

cards_proj/src/cards/cli.py
```
import cards
```

Through this cards namespace, cli.py accesses:

- cards.__version__ (a string)
- cards.CardDB (a class representing the main API methods)
- cards.InvalidCardID (an exception)
- cards.Card (the primary data type for use between the CLI and API)

Most of the API access is through a context manager that creates a cards.CardsDB object:

cards_proj/src/cards/cli.py
```
@contextmanager
def cards_db():
    db_path = get_path()
    db = cards.CardsDB(db_path)
    yield db
    db.close()
```

Then, most of the functions work through that object. For example, the start command accesses db.start() through db, a CardsDB instance:

cards_proj/src/cards/cli.py
```
@app.command()
def start(card_id: int):
    """Set a card state to 'in prog'."""
    with cards_db() as db:
        try:
            db.start(card_id)
        except cards.InvalidCardId:
            print(f"Error: Invalid card id {card_id}")
```

Both add and update also use the cards.Card data structure we've played with before:

cards_proj/src/cards/cli.py
```
db.add_card(cards.Card(summary, owner, state="todo"))
```

And the version command looks up cards.__version__:

cards_proj/src/cards/cli.py
```
@app.command()
def version():
    """Return version of cards application"""
    print(cards.__version__)
```

For the sake of what to mock for testing the CLI, let's mock both __version__ and CardsDB.

The version command looks pretty simple. It just accesses cards.__version__ and prints that. So let's start there.

Testing with Typer

A great feature of Typer is that it proves a testing interface. With it, we can call our application without having to resort to using subprocess.run, which is good, because we can't mock stuff running in a separate process. (We looked at a short example of using subprocess.run with test_version_v1 in Using capsys, on page ?.) We just need to give the runner's invoke function our app—cards.app—and a list of strings that represents the command.

Here's an example of invoking the version function:

ch10/test_typer_testing.py
```python
from typer.testing import CliRunner
from cards.cli import app

runner = CliRunner()

def test_typer_runner():
    result = runner.invoke(app, ["version"])
    print()
    print(f'version: {result.stdout}')

    result = runner.invoke(app, ["list", "-o", "brian"])
    print(f'list:\n{result.stdout}')
```

In the example test:

- To run cards version, we run runner.invoke(app, ["version"]).
- To run cards list -o brian, we run runner.invoke(app, ["list", "-o", "brian"]).

We don't have to include "cards" in the list to send to the app, and the rest of the string is split into a list of strings.

Let's run this code and see what happens:

```
$ pytest -v -s test_typer_testing.py::test_typer_runner
========================= test session starts ==========================
collected 1 item

test_typer_testing.py::test_typer_runner
version: 1.0.0

list:

  ID │ state │ owner │ summary
 ━━━━━━┿━━━━━━━━━┿━━━━━━━━━┿━━━━━━━━━━━━━━━━━━━━━━━━━━━━
```

```
  3  │ todo  │ brian │ Finish second edition
     ╵       ╵       ╵

PASSED
========================= 1 passed in 0.05s ==========================
```

Looks like it works, and is running against the live database.

However, before we move on, let's write a helper function called cards_cli. We know we're going to invoke the app plenty of times during testing the CLI, so let's simplify it a bit:

ch10/test_typer_testing.py
```python
import shlex
def cards_cli(command_string):
    command_list = shlex.split(command_string)
    result = runner.invoke(app, command_list)
    output = result.stdout.rstrip()
    return output

def test_cards_cli():
    result = cards_cli("version")
    print()
    print(f'version: {result}')

    result = cards_cli("list -o brian")
    print(f'list:\n{result}')
```

This allows us to let shlex.split() turn "list -o brian" into ["list", "-o", "brian"] for us, as well as grab the output and return it.

Now we're ready to get back to mocking.

Mocking an Attribute

Most of the Cards API is accessed through a CardsDB object, but one entry point is just an attribute, cards.__version__. Let's look at how we can use mocking to make sure the value from cards.__version__ is correctly reported through the CLI.

The mock package is part of the standard library and is found at unittest.mock. We need to import it before we can use it:

ch10/test_mock.py
```python
from unittest import mock
```

The patch method is the workhorse of the mock package. We'll use it primarily in its context manager form. Here's what it looks like to mock __version__:

ch10/test_mock.py
```python
def test_mock_version():
    with mock.patch('cards.cli.cards') as mock_cards:
        mock_cards.__version__ = '1.2.3'
        result = runner.invoke(app, ["version"])
        assert result.stdout.rstrip() == '1.2.3'
```

We have a couple of new elements here. When the CLI prints stuff, print adds a newline, so we strip that newline off of the output with result.stdout.rstrip(). But that's not the weird part.

The weird part is 'cards.cli.cards'. In order to use mocks successfully, you need to think about patching parts of the system as they are accessed. The cards.cli includes an import cards command, which makes cards part of the cards.cli namespace. It's at that level we are trying to mock things, changing the view of cards from the perspective of cards.cli.

The call to mock.patch() used as a context manager within a with block returns a mock object that is cleaned up after the with block. But inside the block, everything in cards.cli that accesses cards will see a mock object instead. The as mock_cards saves this mock object to the mock_cards variable.

The following line:

mock_cards._version_ == '1,2,3'

sets the _version_ attribute to whatever we want. Now when we invoke the version command, it's this _version_ attribute that is accessed.

Mock is replacing part of our system with something else, namely mock objects. With mock objects, we can do lots of stuff, like setting attribute values, return values for callables, and even look at how callables are called.

If that last bit was confusing, you're not alone. This weirdness is one of the reasons many people avoid mocking altogether. But once you get your head around that, the rest kinda sorta makes sense.

In the upcoming sections, we'll look at mocking classes and methods of classes.