October 19th, 2023

Mocking in Python

Presented by
Ondrej Kinst
okinst@redhat.com

Test Double

Test Double represents an object at which the SUT (and its test) is dependent on.

- Needs to have the same interface (Sometimes, like in Python, not entirely)
- Shouldn’t be dependent on anything else

Different kinds of Test Doubles

Dummy

Exists but doesn't *do* anything

Shim

A detour, replaces library calls

Spy

Watches and logs all interactions

Fake

Simulate a test object through a different implementation

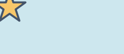
Stub

Returns a predetermined value

Mock

Watches, verifies, returns

Why mocking?



1 Behavior verification

Can test sequence of events or invocations fulfilled by a certain object.

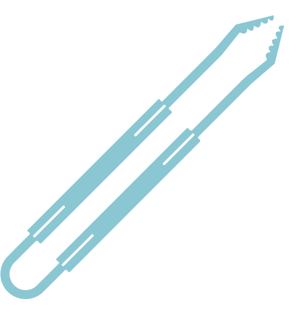
2 Isolate and focus

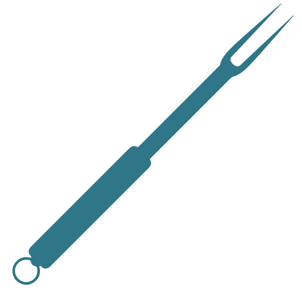
We focus on the SUT, not on the behavior/state of external dependencies.

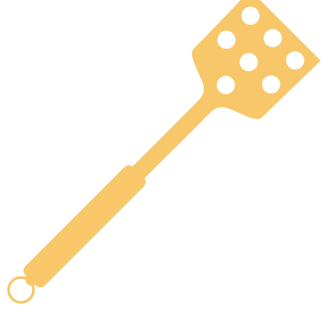
3 Potential speedup

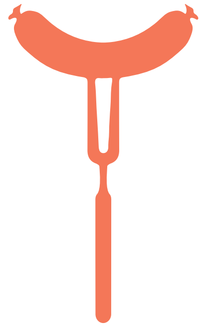
Test execution can even get a bit faster by avoiding database queries or network calls.

The unittest.mock









Part of the standard library

If you're using Python 3.3 and above

Provides the Mock class

As well as the MagicMock subclass

Provides patch()

Can be used as a decorator or context manager to replace real objects in your code.

Provides create_autospec()

Limits the API of the mocks to the API of the original object. Helps fix some of the inherent flaws in mocking.

Using the Mock object

```
In [1]: from unittest.mock import Mock

In [2]: my_mock = Mock(name="Carl")

In [3]: my_mock
Out[3]: <Mock name='Carl' id='4416976464'>

In [4]: my_mock.return_value = "Hello"

In [5]: my_mock()
Out[5]: 'Hello'
```

Using the Mock object

```
In [6]: my_mock.side_effect = Exception

In [7]: my_mock()
---------------------------------------------------------
Exception                       Traceback (most recent call last)
…
In [8]: my_mock.side_effect = [2, 1, 0]

In [9]: my_mock(), my_mock(), my_mock()
Out[9]: (2, 1, 0)

In [10]: my_mock.side_effect = sum

In [10]: my_mock([1, 2, 3])
Out[10]: 6
```

Using the Mock object

```
In [11]: my_mock.foo
Out[11]: <Mock name='Carl.foo' id='4409941328'>

In [12]: my_mock.foo.bar
Out[12]: <Mock name='Carl.foo.bar' id='4410818000'>

In [13]: my_mock.foo.bar.baz
Out[13]: <Mock name='Carl.foo.bar.baz' id='4410702672'>

In [14]: sub_mock = Mock(baz="Hey!")

In [15]: my_mock.foo.bar = sub_mock

In [16]: my_mock.foo.bar.baz
Out[16]: 'Hey!'
```

The verify step

```
assert mock_func.called

mock_func.assert_called_once()

mock_func.assert_called_once_with(1, "asdf”)
```

Using patch() as function decorator

```
class TestCase1(unittest.TestCase):

    @patch('module.os.getcwd')
    @patch('module.SomeFunction')
    @patch('module.SomeClass')
    def test1(self, MockClass, mock_function,
              mock_getcwd):
        mock_function.return_value = "testing"
        …

    @patch.dict('os.environ', {'new_key': 'new_value'})
    def test2(self):
        …
```

Using patch() as context manager

```
some_dict = {}
with patch.dict(some_dict, {'key': 'value'}
                ) as patched_dict:
    assert some_dict == {'key': 'value'}
    assert patched_dict == {'key': 'value'}

assert some_dict == {}
```

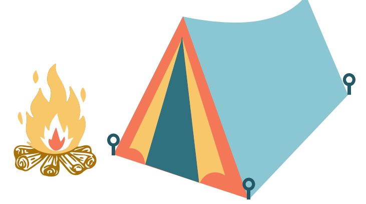
Example

```
@patch('argparse.ArgumentParser.parse_args')
@patch('reporter.settings.QUEUE.consume_messages')
def test_main_listen(self, consume_messages_mock,
                     args_mock):
    """Test main with listen argument."""
    from reporter.__main__ import main

    args_mock.return_value = self.listen_args
    main()
    assert consume_messages_mock.called
```

https://gitlab.com/cki-project/reporter/-/blob/main/tests/test_main.py

Underlying issues



Mock is too flexible
With great power comes great responsibility

```
mock = Mock(return_value=None)

mock(123)

mock.aasert_called_once_with("foo")
```

Testing old API after refactoring

```
class Person:
        def set_name(self, new_name)
                self.name = new_name

def rename(person, name):
        person.set_name(name)
…

def test_rename(self):
        target = Mock()
        rename(target, "Bob")
        target.set_name.\
                assert_called_with("Bob")
```

Counteracting these issues

```
mock = Mock(spec=module.MyClass)


patch(..., autospec=True)

mock_function = create_autospec(function)


seal(mock)
```

More examples with some utility libraries

```
from freezegun import freeze_time
import datetime


@freeze_time("2012-01-14")
def test():
    assert datetime.datetime.now() == datetime.datetime(2012, 1, 14)
```

More examples with some utility libraries

```
import responses
import requests

@responses.activate
def test_404():
    responses.add(
        responses.GET,
        "http://my_app.com/api/1/foo",
        json={"error": "not found"},
        status=404,
    )
    resp = requests.get("http://my_app.com/api/1/foo")

    assert resp.json() == {"error": "not found"}
    assert resp.status_code == 404
```

Conclusion

1 **Don’t overdo it**
Remember that you still need to test some real objects.

2 **Specify your mocks**
To avoid instances where you’re not really testing anything.

3 **Consider your options**
Especially when doing TDD, you can still do state verification if you want to.

THANK YOU

Q&A