Mock, Patch

When and how to use

Maciej Polańczyk

maciej.polanczyk@stxnext.pl

Schedule

- Class to be tested
- Basic test
- Mock
- Not the best mock
- The best mock - spec_set
- Good mock - spec
- Mock requests.post
- Patch.object
- Patch
- Patch - raise Exception
- Patch - different return_value
- Patch - method instead of return_value
- Patch - nesting patches
- Patch - TestCase
- Patch - in setUp
- Patch - environment variables
- Patch - properties

https://docs.python.org/3/library/unittest.mock.html

Class to be tested

```python
import requests

class MessageSender:
  _max_attempts = 3

  def send(self, message):
      data = {
          'from': message.sender,
          'to': message.receiver,
          'subject': message.subject,
      }
      for attempt in range(self._max_attempts):
          response = requests.post(
              url='http://example.com/email',
              data=data,
          )
          if response.status_code == requests.codes.created:
              return True
      return False
```

Basic test

```
from message_sender import MessageSender

class MessageSenderTests(unittest.TestCase):

  def test_should_send_message():
      # GIVEN
      message_sender = MessageSender()
      message_to_be_sent = self._get_mocked_message()
      # WHEN
      result = message_sender.send(message_to_be_sent)
      # THEN
      self.assertTrue(result)
      # assert correct arguments were passed to request.post
```

Mock - creation

```
from unittest.mock import Mock

# mock and attributes
mock = Mock()
print(mock.not_existing_field)
<Mock name='mock.not_existing_field' id='4327526528'>

mock = Mock()
mock.existing_field = 5
print(mock.existing_field)
5

mock = Mock(existing_field=5)
print(mock.existing_field)
5
```

```
from unittest.mock import Mock

# mock and methods
mock = Mock()
print(mock.not_existing_method())
<Mock name='mock.not_existing_method()' id='4372615856'>

mock = Mock()
mock.existing_method.return_value = 5
print(mock.existing_method())
5

mock = Mock(existing_method=Mock(return_value=5))
print(mock.existing_method())
5
```

Mock - assert usage

```
from unittest.mock import Mock

mock = Mock()
mock.existing_method.return_value = 5
mock.existing_method()
mock.existing_method.assert_called_once_with()

mock = Mock()
mock.existing_method.return_value = 5
mock.existing_method(1)
mock.existing_method.assert_called_once_with(1)

mock = Mock()
mock.existing_method.return_value = 5
mock.existing_method(1)
mock.existing_method(2)
mock.existing_method.assert_any_call(1)
```

```
from unittest.mock import Mock, call

mock = Mock()
mock.existing_method.return_value = 5
mock.existing_method(1)
mock.existing_method(2)
print(mock.call_args_list)
[]
print(mock.method_calls)
[call.existing_method(1), call.existing_method(2)]
print(mock.mock_calls)
[call.existing_method(1), call.existing_method(2)]

self.assertEqual(mock.mock_calls, [call.existing_method(1), call.existing_method(2)])
self.assertEqual(mock.existing_method.mock_calls, [call(1), call(2)])
```

Not the best mock

```
from unittest.mock import Mock


@staticmethod
def _get_mocked_message():
  message_mock = Mock(
    receiver='maciej.polanczyk@stxnext.pl',
    sender='maciej.polanczyk@test.com',
    subject='testing sending message'
  )
  return message_mock
```

The best mock - spec_set

```
from unittest.mock import Mock
from message import Message


@staticmethod
def _get_mocked_message():
  message_mock = Mock(
      spec_set=Message,
      receiver='maciej.polanczyk@stxnext.pl',
      sender='maciej.polanczyk@test.com',
      subject='testing sending message'
  )
  return message_mock
```

Good mock - spec

```
import requests
from unittest.mock import Mock

@staticmethod
def _get_success_response():
  response_mock = Mock(
     spec=requests.Response,
     status_code=requests.codes.created
  )
  return response_mock

@staticmethod
def _get_failure_response():
  response_mock = Mock(
     spec=requests.Response,
     status_code=requests.codes.bad
  )
  return response_mock
```

Basic test

```
from message_sender import MessageSender

class MessageSenderTests(unittest.TestCase):

  def test_should_send_message():
     # GIVEN
     message_sender = MessageSender()
     message_to_be_sent = self._get_mocked_message()
     # WHEN
     result = message_sender.send(message_to_be_sent)
     # THEN
     self.assertTrue(result)
     # assert correct arguments were passed to request.post
```

Mock request.post

```
import requests

class MessageSender:
  _max_attempts = 3

  def send(self, message):
      data = {
          'from': message.sender,
          'to': message.receiver,
          'subject': message.subject,
      }
      for attempt in range(self._max_attempts):
          response = requests.post(
              url='http://example.com/email',
              data=data,
          )
          if response.status_code == requests.codes.created:
              return True
      return False
```

Mock request.post

```
class MessageSender:
  _max_attempts = 3
  def __init__():
    self._sender = ...

  def send(self, message):
    data = {
      'from': message.sender,
      'to': message.receiver,
      'subject': message.subject,
    }
    for attempt in range(self._max_attempts):
      response = self._sender.post(
        url='http://example.com/email',
        data=data,
      )
      if response.status_code == requests.codes.created:
        return True
    return False
```

```
class MessageSenderTests(unittest.TestCase):

  def test_should_send_message(self):
    # GIVEN
    message_sender = MessageSender()
    message_sender._sender = self._get_mocked_post_method()
    message_to_be_sent = self._get_mocked_message()
    # WHEN
    result = message_sender.send(message_to_be_sent)
    # THEN
    self.assertTrue(result)

  @staticmethod
  def _get_mocked_post_method():
    post_method_mock = Mock(
      # spec_set=SenderClass
      return_value=MessageSenderTests._get_success_response()
    )
    return post_method_mock
```

Patch.object

```
import requests
from unittest.mock import patch
from message_sender import MessageSender

@patch.object(requests, 'post', autospec=True)
def test_should_send_message(self, post_mock):
  # GIVEN
  post_mock.return_value = self._get_success_response()
  message_sender = MessageSender()
  message_to_be_sent = self._get_example_message()
  # WHEN
  result = message_sender.send(message_to_be_sent)
  # THEN
  self.assertTrue(result)
  post_mock.assert_called_once_with(
     data={
        'from': 'maciej.polanczyk@test.com',
        'subject': 'testing sending message',
        'to': 'maciej.polanczyk@stxnext.pl'
     },
     url='http://example.com/email'
  )
```

Patch

```
from unittest.mock import patch
from message_sender import MessageSender

@patch('requests.post', autospec=True)
def test_should_send_message(self, post_mock):
  # GIVEN
  post_mock.return_value = self._get_success_response()
  message_sender = MessageSender()
  message_to_be_sent = self._get_example_message()
  # WHEN
  result = message_sender.send(message_to_be_sent)
  # THEN
  self.assertTrue(result)
  post_mock.assert_called_once_with(
     data={
        'from': 'maciej.polanczyk@test.com',
        'subject': 'testing sending message',
        'to': 'maciej.polanczyk@stxnext.pl'
     },
     url='http://example.com/email'
  )
```

Patch - raise Exception

```
import requests

class MessageSender:
  _max_attempts = 3

  def send(self, message):
      data = {
          'from': message.sender,
          'to': message.receiver,
          'subject': message.subject,
      }
      for attempt in range(self._max_attempts):
          response = requests.post(
              url='http://example.com/email',
              data=data,
          )
          if response.status_code == requests.codes.created:
              return True
      return False
```

Patch - raise Exception

```
@patch.object(requests, 'post')
def test_should_not_catch_exception(self, post_mock):
  # GIVEN
  post_mock.side_effect = RequestException(
     'Expected exception from unit tests'
  )
  message_sender = MessageSender()
  message_to_be_sent = self._get_example_message()
  # WHEN & THEN
  with self.assertRaisesRegex(RequestException, 'Expected exception'):
     message_sender.send(message_to_be_sent)
  post_mock.assert_called_once_with(
     data={
        'from': 'maciej.polanczyk@test.com',
        'subject': 'testing sending message',
        'to': 'maciej.polanczyk@stxnext.pl'
     },
     url='http://example.com/email'
  )
```

Patch - different return_value

```
import requests

class MessageSender:
  _max_attempts = 3

  def send(self, message):
      data = {
          'from': message.sender,
          'to': message.receiver,
          'subject': message.subject,
      }
      for attempt in range(self._max_attempts):
          response = requests.post(
              url='http://example.com/email',
              data=data,
          )
          if response.status_code == requests.codes.created:
              return True
      return False
```

Patch - different return_value

```
@patch.object(requests, 'post', autospec=True)
def test_should_retry_sending_when_incorrect_status_received(self, post_mock):
  # GIVEN
  post_mock.side_effect = [self._get_failure_response(),
                           self._get_failure_response(),
                           self._get_failure_response(),]
  message_to_be_sent = self._get_example_message()
  message_sender = MessageSender()
  # WHEN
  result = message_sender.send(message_to_be_sent)
  # THEN
  self.assertFalse(result)
  expected_calls = [self._get_expected_call(),
                    self._get_expected_call(),
                    self._get_expected_call(),]
  self.assertEqual(post_mock.call_args_list, expected_calls)
```

```
def _get_expected_call(self):
  return call(
     data = {
        'from': 'maciej.polanczyk@test.com',
        'subject': 'testing sending message',
        'to': 'maciej.polanczyk@stxnext.pl'
     },
     url = 'http://example.com/email'
  )
```

Patch - method instead of return_value

```
@patch.object(requests, 'post', autospec=True)
def test_should_send_message_tooo(self, post_mock):
  # GIVEN
  def implementation_from_unit_test(*args, **kwargs):
     return self._get_success_response()

  post_mock.side_effect = implementation_from_unit_test
  message_to_be_sent = self._get_example_message()
  message_sender = MessageSender()
  # WHEN
  result = message_sender.send(message_to_be_sent)
  # THEN
  self.assertTrue(result)
  post_mock.assert_called_once_with(
     data={
        'from': 'maciej.polanczyk@test.com',
        'subject': 'testing sending message',
        'to': 'maciej.polanczyk@stxnext.pl'
     },
     url='http://example.com/email'
  )
```

Patch - nesting patches

```
@patch.object(requests, 'post', autospec=True)
@patch.object(requests, 'get', autospec=True)
@patch.object(requests, 'put', autospec=True)
def test_should_send_message_tooo(self, put_mock, get_mock, post_mock):
  # GIVEN
  ...
  # WHEN
  ...
  # THEN
  ...
```

Patch - TestCase

```
class MessageSenderTests(unittest.TestCase):

  @patch.object(requests, 'post', autospec=True)
  def test_should_send_message(self, post_mock):
      pass

  @patch.object(requests, 'post', autospec=True)
  def test_should_not_catch_exception(self, post_mock):
      pass

  @patch.object(requests, 'post', autospec=True)
  def test_should_retry_sending_when_incorrect_status_received(self, post_mock):
      pass
```

```
@patch.object(requests, 'post', autospec=True)
class MessageSenderTests(unittest.TestCase):

  def test_should_send_message(self, post_mock):
      pass

  def test_should_not_catch_exception(self, post_mock):
      pass

  def test_should_retry_sending_when_incorrect_status_received(self, post_mock):
      pass
```

Patch - in setUp

```
class MessageSenderTests(unittest.TestCase):

  def setUp(self):
     super().setUp()
     patcher = patch('requests.post')
     self.addCleanup(patcher.stop)
     self._post_mock = patcher.start()

  def test_should_send_message(self):
     # GIVEN
     self._post_mock.return_value = self._get_success_response()
     # WHEN
     ...
     # THEN
     ...
```

Patch - environment variables

```
class MessageSenderTests(unittest.TestCase):


  @patch.dict('os.environ', {'not_existing_key': 'some_value'}, clear=True)
  def test_should_override_environment_variables(self):
     self.assertEqual(os.environ, {'not_existing_key': 'some_value'})
```

Thank you!