Python Mocking

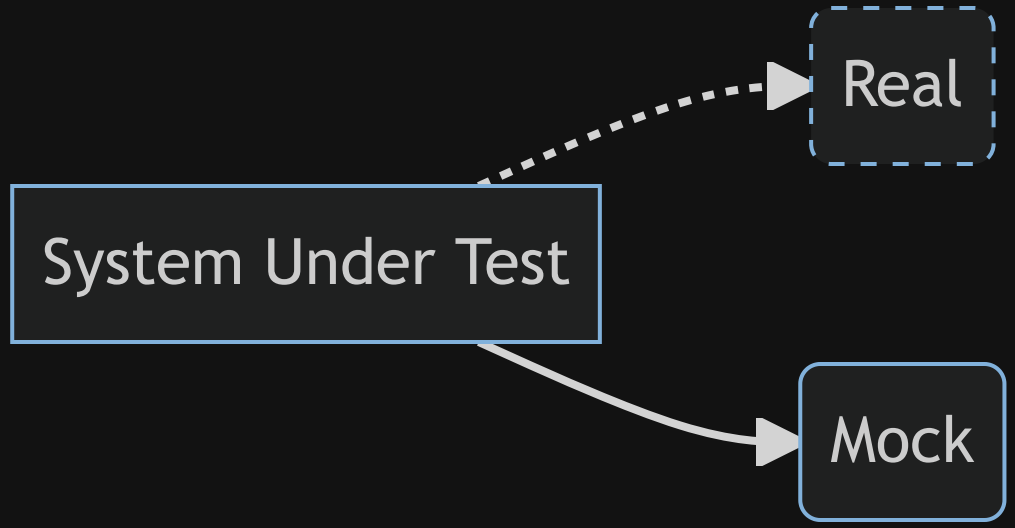
Walkthrough of how to mock in python

KHEMMATAT THEANVANICHPANT (TUI)

Table of Content

What is mocking?

A type of test double



Definition

"Mocks are pre-programmed with expectations which form a specification of the calls they are expected to receive. They can throw an exception if they receive a call they don’t expect and are checked during verification to ensure they got all the calls they were expecting."

Martin Fowler

Python mocking

Import and Namespace

- individual import
- module import

```
1   from openpyxl import Workbook
```

```
1   import requests
```

Where to patch

`patch("path.to.object")`

How to know where to patch

1. How is it imported?
2. Where is it used?

Individual import

```python
# main.py
import string
from magic import get_magic_number

def get_magic_char() -> str:
    chars = (
        string.ascii_letters + string.punctuation
    )
    magic_number = get_magic_number()
    return chars[magic_number % len(chars)]
```

```python
# magic.py
import random

def get_magic_number() -> int:
    return random.randint(1, 100)
```

```python
# test_main.py
from unittest import mock
from main import get_magic_char

def test_get_magic_char() -> None:
    with mock.patch("main.get_magic_number", return_value=2):
        actual = get_magic_char()
        expected = "c"
        assert actual == expected
```

Module import

```python
# main.py
import string
import magic

def get_magic_char() -> str:
    chars = (
        string.ascii_letters + string.punctuation
    )
    magic_number = magic.get_magic_number()
    return chars[magic_number % len(chars)]
```

```python
# magic.py
import random

def get_magic_number() -> int:
    return random.randint(1, 100)
```

```python
# test_main.py
from unittest import mock
from main import get_magic_char

def test_get_magic_char() -> None:
    with mock.patch("magic.get_magic_number", return_value=2):
        actual = get_magic_char()
        expected = "c"
        assert actual == expected
```

Patching

`unittest.mock.patch`

`patch()`

```
patch("path.to.object.attribute")
```

- builtins
- no need to import module

`patch.object()`

```
patch.object(path.to.object, "attribute")
```

- requires importing module to patch first
- make refactoring easier

Patch scope

1. context manager
2. function decorator
3. class decorator
4. inline (need to manually call method to start and stop the mocking)

1. Context manager

```
with patch.object(some_module, "some_function") as mock_some_function:
    ...
```

2. Function decorator

```
@patch.object(some_module, "some_function")
@patch.object(some_module, "another_function")
def test_foo(another_function: mock.MagicMock, some_function: mock.MagicMock) -> None:
    ...
```

3. Class decorator

```
@patch.object(some_module, "some_function")
class TestCase:
    def test_foo(some_function: mock.MagicMock) -> None:
        ...

    def test_bar(some_function: mock.MagicMock) -> None:
        ...
```

4. Inline (need to manually call method to start and stop the mocking)

`pytest`

```
def test_foo(request: pytest.FixtureRequest) -> None:
    patcher = patch.object(some_module, "some_function")
    patcher.start()
    request.addfinalizer(patcher.stop)
    ...
```

`unittest`

```
class TestCase(unittest.TestCase):
    def setUp(self) -> None:
        patcher = patch.object(some_module, "some_function")
        patcher.start()
        self.addCleanup(patcher.stop)

    def test_foo(self) -> None:
        ...
```

`pytest-mock`

```python
# test_main.py
from unittest import mock
from pytest_mock import MockerFixture
import magic
from main import get_magic_char

def test_get_magic_char(mocker: MockerFixture) -> str:
    mocker.patch.object(magic, "get_magic_number", return_value=2)
    actual = get_magic_char()
    expected = "c"
    assert actual == expected
```

kwargs

1. `return_value`
2. `side_effect`
3. `new`

`return_value`

```python
# main.py
import string
import magic

def get_magic_char() -> str:
    chars = (
        string.ascii_letters + string.punctuation
    )
    magic_number = magic.get_magic_number()
    return chars[magic_number % len(chars)]
```

```python
# magic.py
import random

def get_magic_number() -> int:
    return random.randint(1, 100)
```

```python
# test_main.py
from unittest import mock
import magic
from main import get_magic_char

def test_get_magic_char() -> str:
    with mock.patch.object(magic, "get_magic_number", return_value=2):
        actual = get_magic_char()
        expected = "c"
        assert actual == expected
```

`side_effect`

dynamic return value

1. Callable
2. Exception
3. Iterable

`side_effect`: Callable

```python
# main.py
from decimal import Decimal
import db

def get_book_prices(book_names: list[str]) -> dict[str, Decimal]:
    price_map = {}
    for name in book_names:
        book = db.get_book_by_name(book_name)
        price = book.price if book else None
        price_map[name] = date_published
    return price_map
```

```python
# db.py
def get_book_by_name(name: str) -> Optional[Book]:
    return Book.objects.filter(name=name).first()
```

`side_effect`: Callable (continue)

```python
# test_main.py
from unittest import mock
import db
from main import get_book_prices

def test_get_book_prices() -> None:
    name_to_book = {"Foo": Book(price=10)}
    def side_effect(book_name):
        return name_to_book.get(book_name)

    with mock.patch.object(db, "get_book_by_name", side_effect=side_effect):
        actual = get_book_prices(["Foo", "Bar"])
        expected = {
            "Foo": 10,
            "Bar": None,
        }
        assert actual == expected
```

`side_effect`: Exception

```python
# main.py
import openpyxl
from openpyxl.utils.exceptions import InvalidFileException
from rest_framework import serializers

def load_workbook(filename: str) -> openpyxl.Workbook:
    try:
        return openpyxl.load_workbook(filename)
    except InvalidFileException:
        raise serializers.ValidationError("Failed to read an excel file")
```

```python
import pytest
import openpyxl
from openpyxl.utils.exceptions import InvalidFileException
from rest_framework import serializers
from main import load_workbook

def test_load_workbook_with_invalid_file(mocker) -> None
    mocker.patch.object(openpyxl, "load_workbook", side_effect=InvalidFileException)
    with pytest.raises(serializers.ValidationError) as exc_info:
        load_workbook("foo.xlsx")
    expected_error_msg = "Failed to read an excel file"
    actual_error_msg, = exc_info.value.args
    assert actual_error_msg == expected_error_msg
```

`side_effect`: Iterable

```python
# main.py
def get_magic_text(length: int) -> str:
    lookup_chars = string.ascii_letters + string.punctuation
    lookup_chars_len = len(lookup_chars)
    chars = []
    for _ in range(length):
        magic_num = magic.get_magic_number()
        chars.append(lookup_chars[magic_num % lookup_chars_len])
    return "".join(chars)
```

```python
# test_main.py
def test_get_magic_text() -> None
    with mock.patch.object(magic, "get_magic_number", side_effect=[2, 0]):
        actual = get_magic_text(2)
        expected = "ca"
        assert actual == expected
```

`new`

patch attribute

```python
# utils.py
import constants

def generate_apps_markdown_entry() -> str:
    entries = ["Apps\n"]
    for app in constants.APPS:
        entries.append(f"- {app}\n")
    return "".join(entries)
```

```python
# constants.py
APPS = [
    "core",
    "master_data",
    "site_settings",
    "drawing",
    "visualization",
    "planning",
    "evaluation",
    "summary",
    "note",
    "management",
    "users",
    "tracking",
    "issues",
]
```

`new` (continue)

```
import textwrap
import constants
from utils import generate_apps_markdown_entry

def test_generate_apps_markdown_entry() -> None:
    # \ to start with the string and avoid the first newline
    expected = textwrap.dedent(
        """\
        Apps
        - foo
        - bar
        """
    )

    with mock.patch.object(constants, "A", new=["foo", "bar"]):
        actual = generate_apps_markdown_entry()
        assert actual == expected
```

Mock class

Mock class

- create new attribute on the fly when they are accessed

```
mock = Mock()
mock.filter(name="foo").order_by("-price")
```

- record how it is called

```
>>> mock.mock_calls
[call.filter(name='foo'), call.filter().order_by('-price')]
```

- assert how it is called

```
mock.filter.assert_called_once_with(name="foo")
mock.filter.return_value.order_by.assert_called_once_with("-price")
```

`Mock` vs `MagicMock`

`Mock`

- Can set `return_value`
- Can set `side_effect`
- Has `assert_*` methods

`MagicMock`

- subclass of `Mock`
- implemented some magic methods such as
  - `__len__()`
  - `__bool__()`
  - etc...

Debugging

`Mock.mock_calls`

Speccing

Pitfalls of mock

Performing the followings on mock object will not result in an error

1. access non existing attributes or calling non existing methods
2. attemp to set non existing attributes
3. calling methods or functions with wrong/missing arguments

`spec`

raise an error when accessing non existing attributes/methods

`Mock` or `MagicMock`

```
from restframework.request import Request

mock_request = mock.MagicMock(spec=Request)
```

`patch()` or `patch.object()`

```

with mock.patch("some_module.Request", spec=True):
    ...
```

`spec_set`

raise an error when accessing non existing attributes/methods or setting non existing attributes

`Mock` or `MagicMock`

```
from restframework.request import Request

mock_request = mock.MagicMock(spec_set=Request)
```

`patch()` or `patch.object()`

```

with mock.patch("some_module.Request", spec_set=True):
    ...
```

`autospec`

like `spec` but also raise an error when accessing attribute of attribute or using invalid method arguments

`Mock` or `MagicMock`

```
from restframework.request import Request

mock_request = mock.create_autospec(Request)
```

`patch()` or `patch.object()`

```

with mock.patch("some_module.Request", autospec=True):
    ...
```

`spec` and `autospec` limitations

- Do not know about dynamic attributes (attributes set via constructors also count as dynamic)
- `autospec` has to access attributes of the real object and may trigger some code execution

For more details, read the official Autospeccing docs

Limitation of mock

Global variable with side effect

```
foo = some_function()
```

Class variable with side effect

```
class Config:
    bar = some_function()
```

Libraries

Libraries

- `freezegun` or `time-machine` for mocking builtins datetime module
- `responses` for mocking `requests` library

Resources

Official docs

- https://docs.python.org/3/library/unittest.mock.html#where-to-patch
- https://docs.python.org/3/library/unittest.mock-examples.html
- https://docs.python.org/3/library/unittest.mock.html#autospeccing

Video

- PyCon: Demystifying the Patch Function

Blog

- https://alpopkes.com/posts/python/mocking/#how-to-mock