What the mock? — A cheatsheet for mocking in Python

Yeray Diaz · Follow

10 min read · May 9, 2018

Listen Share More

It's just a fact of life, as code grows eventually you will need to start adding mocks to your test suite. What started as a cute little two class project is now talking to external services and you cannot test it comfortably anymore.

That's why Python ships with `unittest.mock`, a powerful part of the standard library for stubbing dependencies and mocking side effects.

However, `unittest.mock` is not particularly intuitive.

I've found myself many times wondering why my go-to recipe does not work for a particular case, so I've put together this cheatsheet to help myself and others get mocks working quickly.

You can find the code examples in the article's Github repository. I'll be using Python 3.6, if you're using 3.2 or below you'll need to use the `mock` PyPI package.

The examples are written using `unittest.TestCase` classes for simplicity in executing them without dependencies, but you could write them as functions using `pytest` almost directly, `unittest.mock` will work just fine. If you are a `pytest` user though I encourage you to have a look at the excellent `pytest-mock` library.

The Mock class in a nutshell

The centerpoint of the `unittest.mock` module is, of course, the `Mock` class. The main characteristic of a `Mock` object is that it will return another `Mock` instance when:

- accessing one of its attributes
- calling the object itself

```
1  from unittest import mock
2
3  m = mock.Mock()
4  assert isinstance(m.foo, mock.Mock)
5  assert isinstance(m.bar, mock.Mock)
6  assert isinstance(m(), mock.Mock)
7  assert m.foo is not m.bar is not m()
```

wtmock_01.py hosted with ❤ by GitHub view raw

This is the default behaviour, but it can be overridden in different ways. For example you can assign a value to an **attribute** in the `Mock` by:

- Assign it directly, like you'd do with any Python object.
- Use the `configure_mock` method on an instance.
- Or pass keyword arguments to the `Mock` class on creation.

```
1  m.foo = 'bar'
2  assert m.foo == 'bar'
3
4  m.configure_mock(bar='baz')
5  assert m.bar == 'baz'
```

wtmock_02.py hosted with ❤ by GitHub view raw

To override **calls** to the mock you'll need to configure its `return_value` property, also available as a keyword argument in the `Mock` initializer. The `Mock` will **always return the same value on all calls**, this, again, can also be configured by using the `side_effect` attribute:

- if you'd like to **return different values on each call** you can assign an iterable to `side_effect`.
- If you'd like to **raise an exception** when calling the Mock you can simply assign the exception object to `side_effect`.

```
m.return_value = 42
assert m() == 42

m.side_effect = ['foo', 'bar', 'baz']
assert m() == 'foo'
assert m() == 'bar'
assert m() == 'baz'
try:
    m()
except StopIteration:
    assert True
else:
    assert False

m.side_effect = RuntimeError('Boom')
try:
    m()
except RuntimeError:
    assert True
else:
    assert False
```

wtmock_03.py hosted with ❤ by GitHub view raw

With all these tools we can now create stubs for essentially any Python object, which will work great for inputs to our system. But what about outputs?

If you're making a CLI program to *download the whole Internet*, you probably don't want to download the whole Internet on each test. Instead it would be enough to assert that `requests.download_internet` (not a real method) was called appropriately. `Mock` gives you handy methods to do so.

```python
m.assert_called()
try:
    m.assert_called_once()
except AssertionError:
    assert True
else:
    assert False
```

wtmock_04.py hosted with ❤ by GitHub view raw

Note in our example `assert_called_once` failed, this showcases another key aspect of `Mock` objects, they record **all** interactions with them and you can then inspect these interactions.

For example you can use `call_count` to retrieve the number of calls to the `Mock`, and use `call_args` or `call_args_list` to inspect the arguments to the last or all calls respectively.

If this is inconvenient at any point you can use the `reset_mock` method to clear the recorded interactions, note the configuration will not be reset, just the interactions.

```python
try:
    m(1, foo='bar')
except RuntimeError:
    assert True
else:
    assert False
assert m.call_args == mock.call(1, foo='bar')
assert len(m.call_args_list) > 1

m.reset_mock()
assert m.call_args is None
```

wtmock_05.py hosted with ❤ by GitHub view raw

Finally, let me introduce `MagicMock`, a subclass of `Mock` that implements default *magic* or *dunder* methods. This makes `MagicMock` ideal to mock class behaviour, which is why it's the default class when patching.

We are now ready to start mocking and isolating the unit under tests. Here are a few recipes to keep in mind:

Patch on import

The main way to use `unittest.mock` is to **patch imports in the module under test** using the `patch` function.

`patch` will intercept `import` statements identified by a string (more on that later), and return a `Mock` instance you can preconfigure using the techniques we discussed above.

Imagine we want to test this very simple function:

```
1  import os
2
3  def work_on():
4      path = os.getcwd()
5      print(f'Working on {path}')
6      return path
```

wtmock_01_work.py hosted with ❤ by GitHub view raw

Note we're importing `os` and calling `getcwd` to get the current working directory. We don't want to actually call it on our tests though since it's not important to our code and the return value might differ between environments it run on.

As mentioned above we need to supply `patch` with a string representing our specific import. We do not want to supply simply `os.getcwd` since that would patch it for all modules, instead we want to supply the module under test's `import` of `os`, i.e. `work.os`. When the module is imported `patch` will work its magic and return a `Mock` instead.

```
1  from unittest import TestCase, mock
2
3  from work import work_on
4
5
6  class TestWorkMockingModule(TestCase):
7
8      def test_using_context_manager(self):
9          with mock.patch('work.os') as mocked_os:
10             work_on()
11             mocked_os.getcwd.assert_called_once()
```

wtmock_01_test_work_01.py hosted with ❤ by GitHub view raw

Alternatively, we can use the decorator version of `patch`, note this time the test has an extra parameter: `mocked_os` to which the `Mock` is injected into the test.

```
@mock.patch('work.os')
def test_using_decorator(self, mocked_os):
    work_on()
    mocked_os.getcwd.assert_called_once()
```

wtmock_01_test_work_02.py hosted with ❤ by GitHub view raw

`patch` will forward keyword arguments to the `Mock` class, so to configure a `return_value` we simply add it as one:

```
def test_using_return_value(self):
    """Note 'as' in the context manager is optional"""
    with mock.patch('work.os.getcwd', return_value='testing'):
        assert work_on() == 'testing'
```

wtmock_01_test_work_03.py hosted with ❤ by GitHub view raw

Mocking classes

It's quite common to patch classes completely or partially. Imagine the following simple module:

```python
import os


class Helper:

    def __init__(self, path):
        self.path = path

    def get_path(self):
        base_path = os.getcwd()
        return os.path.join(base_path, self.path)


class Worker:

    def __init__(self):
        self.helper = Helper('db')

    def work(self):
        path = self.helper.get_path()
        print(f'Working on {path}')
        return path
```

wtmock_02_worker.py hosted with ❤ by GitHub view raw

Now there's two things we need to test:

1. `Worker` calls `Helper` with `"db"`
2. `Worker` returns the expected path supplied by `Helper`

In order to test `Worker` in complete isolation we need to patch the whole `Helper` class:

```python
from unittest import TestCase, mock

from worker import Worker, Helper


class TestWorkerModule(TestCase):

    def test_patching_class(self):
        with mock.patch('worker.Helper') as MockHelper:
            MockHelper.return_value.get_path.return_value = 'testing'
            worker = Worker()
            MockHelper.assert_called_once_with('db')
            self.assertEqual(worker.work(), 'testing')
```

wtmock_02_test_worker_01.py hosted with ❤ by GitHub view raw

Note the double `return_value` in the example, simply using `MockHelper.get_path.return_value` would not work since in the code we call `get_path` on an instance, not the class itself.

The chaining syntax is slightly confusing but remember `MagicMock` returns another `MagicMock` on calls `__init__`. Here we're configuring any fake `Helper` instances created by `MockHelper` to return what we expect on calls to `get_path` which is the only method we care about in our test.

Class speccing

A consequence of the flexibility of `Mock` is that once we've mocked a class Python will not raise `AttributeError` as it simply will return new instances of `MagicMock` for basically everything. This is usually a good thing but can lead to some confusing behaviour and potentially bugs. For instance writing the following test,

```python
    def test_patching_class_with_typo(self):
        with mock.patch('worker.Helper') as MockHelper:
            MockHelper.return_value.get_path.return_value = 'testing'
            worker = Worker()
            MockHelper.assrt_called_once_with('db')  # erm....
            self.assertEqual(worker.work(), 'testing')
```

wtmock_02_test_worker_typo.py hosted with ❤ by GitHub view raw

will silently pass with no warning completely missing the typo in `assrt_called_once`.

Additionally, if we were to rename `Helper.get_path` to `Helper.get_folder`, but forget to update the call in `Worker` our tests will still pass:

```python
import os


class Helper:

    def __init__(self, path):
        self.path = path

    def get_folder(self):
        base_path = os.getcwd()
        return os.path.join(base_path, self.path)


class Worker:

    def __init__(self):
        self.helper = Helper('db')

    def work(self):
        path = self.helper.get_path()
        print(f'Working on {path}')
        return path
```

wtmock_03_spec_worker.py hosted with ❤ by GitHub view raw

```python
from unittest import TestCase, mock

from worker import Worker, Helper


class TestWorker(TestCase):

    def test_patching_class(self):
        # this test will give a false positive,
        # there is not `get_path` method but we've mocked it
        with mock.patch('worker.Helper') as MockHelper:
            MockHelper.return_value.get_path.return_value = 'testing'
            worker = Worker()
            MockHelper.assert_called_once_with('db')
            self.assertEqual(worker.work(), 'testing')
```

wtmock_03_spec_false_pass.py hosted with ❤ by GitHub view raw

Luckily `Mock` comes with a tool to prevent these errors, **speccing**.

Put simply, it preconfigures mocks to only respond to methods that actually exist in the spec class. There are several ways to define specs, but the easiest is to simply pass `autospec=True` to the `patch` call, which will configure the `Mock` to behave as the object being mocked, raising exceptions for missing attributes and incorrect signatures as required. For example:

```
    def test_patching_class_with_spec(self):
        with mock.patch('worker.Helper', autospec=True) as MockHelper:
            # the following would raise attribute error
            # MockHelper.return_value.get_path.return_value = 'testing'
            MockHelper.return_value.get_folder.return_value = 'testing'
            worker = Worker()
            MockHelper.assert_called_once_with('db')
            # this test will fail since we we're still using `get_path`
            self.assertEqual(worker.work(), 'testing')
```

wtmock_03_spec_autospec.py hosted with ❤ by GitHub view raw

Partial class mocking

If you're less inclined to testing in complete isolation you can also partially patch a class using `patch.object` :

```
from unittest import TestCase, mock

from worker import Worker, Helper


class TestWorker(TestCase):

    def test_partial_patching(self):
        with mock.patch.object(Helper, 'get_path', return_value='testing'):
            worker = Worker()
            self.assertEqual(worker.helper.path, 'db')
            self.assertEqual(worker.work(), 'testing')
```

wtmock_02_test_worker_partial.py hosted with ❤ by GitHub view raw

Here `patch.object` is allowing us to configure a mocked version of `get_path` only, leaving the rest of the behaviour untouched. Of course this means the test is no longer what you would strictly consider a *unit test* but you may be ok with that.

Mocking built-in functions and environment variables

In the previous examples we neglected to test one particular aspect of our simple class, the `print` call. If in the context of your application this is important, like a CLI command for instance, we need to make assertions against this call.

`print` is, of course a built-in function in Python, which means we do not need to import it in our module, which goes against what we discussed above about patching on import. Still imagine we had this slightly more complicated version of our function:

```
import os


def work_on_env():
    path = os.path.join(os.getcwd(), os.environ['MY_VAR'])
    print(f'Working on {path}')
    return path
```

wtmock_04_builtins.py hosted with ❤ by GitHub view raw

We can write a test like the following:

```
from unittest import TestCase, mock

from worker import work_on_env

class TestBuiltin(TestCase):

    def test_patch_dict(self):
        with mock.patch('worker.print') as mock_print:
            with mock.patch('os.getcwd', return_value='/home/'):
                with mock.patch.dict('os.environ', {'MY_VAR': 'testing'}):
                    self.assertEqual(work_on_env(), '/home/testing')
                    mock_print.assert_called_once_with('Working on /home/testing')
```

wtmock_04_builtins_test.py hosted with ❤ by GitHub view raw

Note a few things:

1. we can mock `print` with no problem and asserting there was a call by following the “patch on import” rule. This however was a change introduced in 3.5, previously you needed to add `create=True` to the `patch` call to signal `unittest.mock` to create a `Mock` even though no import matches the identifier.

2. we're using `patch.dict` to inject a temporary environment variable in `os.environ`, this is extensible to any other dictionary we'd like to mock.

3. we're nesting several `patch` context manager calls but only using `as` in the first one since it's the one we need to use to call `assert_called_once_with`.

If you're not fond of nesting context managers you can also write the `patch` calls in the decorator form:

```
1       @mock.patch('os.getcwd', return_value='/home/')
2       @mock.patch('worker.print')
3       @mock.patch.dict('os.environ', {'MY_VAR': 'testing'})
4       def test_patch_builtin_dict_decorators(self, mock_print, mock_getcwd):
5           self.assertEqual(work_on_env(), '/home/testing')
6           mock_print.assert_called_once_with('Working on /home/testing')
```

wtmock_04_builtins_decorators.py hosted with ❤ by GitHub view raw

Note however the order of the arguments to the test matches the stacking order of the decorators, and also that `patch.dict` does not inject an argument.

Mocking context managers

Context managers are incredibly common and useful in Python, but their actual mechanics make them slightly awkward to mock, imagine this very common scenario:

```
1   def size_of():
2       with open('text.txt') as f:
3           contents = f.read()
4
5       return len(contents)
```

wtmock_05_context_manager_01.py hosted with ❤ by GitHub view raw

You could, of course, add a actual fixture file, but in real world cases it might not be an option, instead we can mock the context manager's output to be a `StringIO` object:

```python
from io import StringIO
from unittest import TestCase, mock

from worker import size_of

class TestContextManager(TestCase):

    def test_context_manager(self):
        with mock.patch('worker.open') as mock_open:
            mock_open.return_value.__enter__.return_value = StringIO('testing')
            self.assertEqual(size_of(), 7)
```

wtmock_05_context_manager_02.py hosted with ❤ by GitHub view raw

There's nothing special here except the magic `__enter__` method, we just have to remember the underlying mechanics of context managers and some clever use of our trusty `MagicMock` .

Mocking class attributes

There are many ways in which to achieve this but some are more fool-proof that others. Say you've written the following code:

```python
class Pricer:

    DISCOUNT = 0.80

    def get_discounted_price(self, price):
        return price * self.DISCOUNT
```

wtmock_06_class_attributes_01.py hosted with ❤ by GitHub view raw

You could test the code without any mocks in two ways:

1. If the code under test accesses the attribute via `self.ATTRIBUTE` , which is the case in this example, you can simply set the attribute directly in the instance. This is fairly safe as the change is limited to this single instance.
2. Alternatively you can also set the attribute in the imported class in the test before creating an instance. This however changes the class attribute you've imported in your test which could affect the following tests, so you'd need to remember to reset it.

The main drawback from this non-Mock approaches is that if you at any point *rename the attribute* your tests will fail and the error will not directly point out this naming mismatch.

To solve that we can make use of `patch.object` on the imported class which will complain if the class does not have the specified attribute.

Here are the some tests using each technique:

```python
from unittest import TestCase, mock, expectedFailure

from pricer import Pricer


class TestClassAttribute(TestCase):

    def test_patch_instance_attribute(self):
        pricer = Pricer()
        pricer.DISCOUNT = 0.5
        self.assertAlmostEqual(pricer.get_discounted_price(100), 50.0)

    def test_set_class_attribute(self):
        Pricer.DISCOUNT = 0.75
        pricer = Pricer()
        self.assertAlmostEqual(pricer.get_discounted_price(100), 75.0)

    @expectedFailure
    def test_patch_incorrect_class_attribute(self):
        with mock.patch.object(Pricer, 'PERCENTAGE', 1):
            pricer = Pricer()
            self.assertAlmostEqual(pricer.get_discounted_price(100), 100)

    def test_patch_class_attribute(self):
        with mock.patch.object(Pricer, 'DISCOUNT', 1):
            pricer = Pricer()
            self.assertAlmostEqual(pricer.get_discounted_price(100), 100)

        self.assertAlmostEqual(pricer.get_discounted_price(100), 80)
```

wtmock_06_class_attributes_02.py hosted with ❤ by GitHub view raw

Mocking class helpers

The following example is the root of many problems regarding monkeypatching using Mock. It usually shows up on more mature codebases that start making use of

frameworks and helpers at class definition. For example, imagine this hypothetical `Field` class helper:

```python
class Field:
    def __init__(self, type_, default, value=None):
        self.type_ = type_
        self.default = default
        self._value = value

    @property
    def value(self):
        if self._value is None:
            return self.default
        else:
            return self._value
```

wtmock_07_class_helpers_01.py hosted with ❤ by GitHub view raw

Its purpose is to wrap and enhance an attribute in another class, a fairly pattern you commonly might see in ORMs or form libraries. Don't concern yourself too much with the details of it just note that there is a `type` and `default` value passed in.

Now take this other sample of production code:

```python
from helper import Field

COUNTRIES = ('US', 'CN', 'JP', 'DE', 'ES', 'FR', 'NL')


class CountryPricer:

    DISCOUNT = 0.8
    country = Field(type_="str", default=COUNTRIES[0])

    def get_discounted_price(self, price, country):
        if country == self.country.value:
            return price * self.DISCOUNT
        else:
            return price
```

wtmock_07_class_helpers_02.py hosted with ❤ by GitHub view raw

This class makes use of the `Field` class by defining its `country` attribute as one whose type is `str` and a `default` as the first element of the `COUNTRIES` constant. The

logic under test is that depending on the country a discount is applied.

For which we might write the following test:

```python
from unittest import TestCase, mock, expectedFailure
from pricer import CountryPricer


class TestCountryPrices(TestCase):

    def test_patch_constant(self):
        with mock.patch('pricer.COUNTRIES', ['GB']):
            pricer = CountryPricer()
            self.assertAlmostEqual(pricer.get_discounted_price(100, 'GB'), 80)  # FAIL
```

wtmock_07_class_helpers_03.py hosted with ❤ by GitHub view raw

But that would NOT pass.

Let's walk through the test:

1. First it patches the default countries in `pricer` to be a list with a single entry `GB` ,
2. This should make the `CountryPricer.country` attribute default to that entry since it's definition includes `default=COUNTRIES[0]` ,
3. It then instanciates the `CountryPricer` and asks for the discounted price for `GB` !

So what's going on?

The root cause of this lies in Python's behaviour during import, best described in Luciano Ramalho's excellent *Fluent Python* on chapter 21*:*

> *For classes, the story is different: at import time, the interpreter executes the body of every class, even the body of classes nested in other classes. Execution of a class body means that the attributes and methods of the class are defined, and then the class object itself is built.*

Applying this to our example:

1. the `country` attribute's `Field` instance is built **before the test is ran** at *import time,*

2. Python reads the body of the class, passing the `COUNTRIES[0]` that is defined at that point to the `Field` instance,

3. Our test code runs but it's too late for us to patch `COUNTRIES` and get a proper assertion.

From the above description you might try delaying the importing of the module until inside the tests, something like:

```
from unittest import TestCase, mock, expectedFailure
# not importing `pricer`

class TestCountryPrices(TestCase):

    def test_delayed_import(self):
        with mock.patch('pricer.COUNTRIES', ['GB']):
            from pricer import CountryPricer
            pricer = CountryPricer()
            self.assertAlmostEqual(pricer.get_discounted_price(100, 'GB'), 80)  # Stil
```

wtmock_07_class_helpers_05.py hosted with ❤ by GitHub view raw

This, however, will still not pass as `mock.patch` will import the module and then monkeypatch it, resulting in the same situation as before.

To work around this we need to embrace the state of the `CountryPricer` class at the time of the test and patch `default` on the already initialized `Field` instance:

```
from unittest import TestCase, mock, expectedFailure
from pricer import CountryPricer


class TestCountryPrices(TestCase):

    def test_patch_class_helper(self):
        with mock.patch('pricer.CountryPricer.country.default', 'GB'):
            pricer = CountryPricer()
            self.assertAlmostEqual(pricer.get_discounted_price(100, 'GB'), 80)

        pricer = CountryPricer()
        self.assertAlmostEqual(pricer.get_discounted_price(100, 'GB'), 100)
```

wtmock_06_class_attributes_04.py hosted with ❤ by GitHub view raw

This solution is not ideal since it requires knowledge of the internals of `Field` which you may not have or want to use if you're using an external library but it works on this admittedly simplified case.

This import time issue is one of the main problems I've encountered while using `unittest.mock`. You need to remember while using it that at import time code at the top-level of modules is executed, including class bodies.

If the logic you're testing is dependent on any of this logic you may need to rethink how you are using `patch` accordingly.

Wrapping up

This introduction and cheatsheet should get you mocking with `unittest.mock`, but I encourage you to read the documentation thoroughly, there's plenty of more interesting tricks to learn.

It's worth mentioning that there are alternatives to `unittest.mock`, in particular Alex Gaynor's `pretend` library in combination with `pytest`'s `monkeypatch` fixture. As Alex points out, using these two libraries you can take a different approach to make mocking stricter yet more predictable. Definitely an approach worth exploring but outside the scope of this article.

`unittest.mock` is currently the standard for mocking in Python and you'll find it in virtually every codebase. Having a solid understanding of it will help you write better tests quicker.

🎉 *Thanks for reading! If you've enjoyed this article you may want to have a look at my series on Python's new built-in concurrency library, AsyncIO:*

- *AsyncIO for the Working Python Developer,*
- *AsyncIO Coroutine Patterns: Beyond await*
- *and AsyncIO Coroutine Patterns: Errors and Cancellation.*

Enjoy! 👋