typemock

Release 0.5.1

Jun 22, 2019

Contents

1 A quick intro **3**
- 1.1 Mock and Verify . . . 3
- 1.2 Type safety . . . 4

2 Installation **5**

3 Type Safety **7**
- 3.1 Strict . . . 8
- 3.2 No return is None return . . . 8
- 3.3 Relaxed . . . 9
- 3.4 During mocking . . . 9

4 Mocking an Object **11**
- 4.1 Mocking Methods . . . 12
  - 4.1.1 Simple response . . . 12
  - 4.1.2 Different responses for different args . . . 12
  - 4.1.3 Series of responses . . . 12
  - 4.1.4 Programmatic response . . . 13
  - 4.1.5 Error responses . . . 13
  - 4.1.6 Arg Matching . . . 13
  - 4.1.7 Mocking async methods . . . 14
- 4.2 Mocking Attributes . . . 14
  - 4.2.1 Simple Get . . . 15
  - 4.2.2 Get Many . . . 15
  - 4.2.3 Get Raise . . . 16
  - 4.2.4 Get programmatic response . . . 16

5 Verifying **17**
- 5.1 Verifying Methods . . . 17
  - 5.1.1 At least once . . . 17
  - 5.1.2 Exactly . . . 17
  - 5.1.3 Never called . . . 18
  - 5.1.4 Any calls . . . 18
- 5.2 Verifying Attributes . . . 18
  - 5.2.1 Get called at least once . . . 18
  - 5.2.2 Get called exact amount of times . . . 19
  - 5.2.3 Set called with specific arg . . . 19

5.2.4 Set called with any arg 19
5.2.5 Set called exact amount of times 19

Warning: This library is still in Alpha. API and implementation could change, and functionality is also not complete.

The mocking tools in python are powerful, flexible and useful for building independent tests at various levels.

This flexibility is part of what is considered a strength of the python language, and possibly any dynamically typed language.

However, this flexibility comes at a cost. Type flexibility in particular.

It is possible to build mocks which do not conform to the actual behaviour or contract defined by the things they are mocking. Or, for them to be initially correct, and then to go out of sync with actual behaviour and for tests to remain green.

We do not have compile time protections for us doing things with/to things which do not align with the contracts they define and the clients of those contracts expect.

But, now we have type hints. And so, we can explicitly define the contracts of our objects, and, if we have done this, we can mock them in a type safe way as well. This is what this library aims to help achieve. Type safe mocking.

Used in conjunction with mypy, this should result in much more high fidelity independent tests.

Note: typemock uses the *typeguard* library to do runtime type checking. Just wanted to give them credit, and make users of this library aware.

CHAPTER 1

A quick intro

Given some class (the implementation of its method is not relevant)

```
class MyThing:

    def multiple_arg(self, prefix: str, number: int) -> str:
        pass
```

1.1 Mock and Verify

We con mock behaviour and verify interactions as follows:

```
from typemock import tmock, when, verify

expected_result = "a string"

with tmock(MyThing) as my_thing_mock:
    when(my_thing_mock.multiple_arg("p", 1)).then_return(expected_result)

actual = my_thing_mock.multiple_arg(
    number=1,
    prefix="p"
)

assert expected_result == actual
verify(my_thing_mock).multiple_arg("p", 1)
```

Things to note:

- The mocked object must be used as a context manager in order to specify behaviour.
- You must let the context close in order to use the defined behaviour.

1.2 Type safety

And when we try to specify behaviour that does not conform to the contract of the object we are mocking

```
expected_result = "a string"

with tmock(MyThing) as my_thing_mock:
    when(my_thing_mock.multiple_arg(prefix="p", number="should be an int")).then_
↪return(expected_result)
```

We get an informative error such as:

```
typemock.safety.MockTypeSafetyError: Method: multiple_arg Arg: number must be of type:
↪<class 'int'>
```

Things to note:

- You will also get type hint errors if you attempt to specify behaviour that returns the incorrect type.

CHAPTER 2

Installation

You can install from pypi:

```
pip install typemock
```

CHAPTER 3

Type Safety

The introduction of type hints to python is great.

Some of the benefits include:

- Clear documentation of the interface of a module, object, function.
- Better IDE tooling. You can drill into the code and investigate how things all wire up when things are explicit.
- Depending on the completeness of your type hints, a reduction in certain types of bugs, and less tests needed to try and avoid them in the first place.

Mypy is an excellent tool for really getting the most out of type hints, and we think it would be great if every project ran the mypy linter on strict mode :D - at least for the public parts of their api.

With a full type hinting, and type safe mocking, the fidelity of your independent internal unit tests can drastically improve.

Also, consider a fully type hinted client to some 3rd party service. A service that does not provide some sort of sandbox or dockerised instance to test against? With type safe mocking of that client, you can now achieve high fidelity tests in this case as well.

We are not suggesting this as a replacement for some sort of integration test if it is possible, but it offers big improvements for a quick lightweight test that can be run independently.

Note: It is also worth mentioning, that running the mypy linter over your type hinted code is also definitely recommended. Regardless of using typemock for your tests. This is key to making sure that your implementations conform to the interface/contract they claim to implement.

Because typemock is aiming to improve type safety it operates in a *strict* mode by default. This section will describe what that means, what other modes are available and how they work in practice.

3.1 Strict

```
TypeSafety.STRICT
```

Strict mode means that when you try to mock a given class, and that class is not fully type hinted, you will get an error. This is the default.

The error will highlight what type hints are missing from the class you are trying to mock. If the class is in your codebase, you can then add them, or if you do not have control over that class, you can look at the other modes for easing up on the type safety.

Lets look at an example.

```
class ClassWithMultipleUnHintedThings:

    def _some_private_function(self):
        # We do not care about type hints for private methods
        pass

    def good_method_with_args_and_return(self, number: int) -> str:
        pass

    def good_method_with_no_args_and_return(self) -> str:
        pass

    def method_with_missing_arg_hint(self, something, something_else: bool) -> None:
        pass

    def method_with_missing_return_type(self):
        pass


with tmock(ClassWithMultipleUnHintedThings) as my_mock: # <- MissingTypeHintsError␣
↪here.
    # Set up mocked behaviour here
    pass
```

With this class, there are multiple missing type hints. And when we try to mock it with the default strict mode, we will get error output as follows:

```
typemock.api.MissingTypeHintsError: ("<class 'test_safety.
↪ClassWithMultipleUnHintedThings'> has missing type hints.", [MissingHint(path=[
↪'method_with_missing_arg_hint', 'something'], member_type=arg), MissingHint(path=[
↪'method_with_missing_return_type'], member_type=return)])
```

We can see that we are missing argument and return type hints. We should try to add those type hints if we can, but if we cannot, we can look at the other type safety modes.

3.2 No return is None return

```
TypeSafety.NO_RETURN_IS_NONE_RETURN
```

This mode lets us be lenient towards methods which do not define a return type. It does however assume that an undefined return type is a return type of *None*.

Here is an example.

```
class NoReturnTypes:

    def method_with_missing_return_type(self):
        pass


with tmock(NoReturnTypes, type_safety=TypeSafety.NO_RETURN_IS_NONE_RETURN) as my_mock:
    when(my_mock.method_with_missing_return_type()).then_return(None)
```

This will no longer raise a *MissingTypeHintsError*. If there were missing argument hints though, it would.

3.3 Relaxed

```
TypeSafety.RELAXED
```

This is the most permissive of the type safety modes. It will allow for a completely unhinted class to be mocked. Obviously many of the benefits of type hinting and type safe mocking are lost in this case.

3.4 During mocking

Typemock also offers type safety at the point at which you specify the behaviour of your mock. And this is probably the most crucial part of it.

If the class you are mocking is type hinted, you cannot make it accept arguments which do not conform to the types expected, and you cannot make the methods return something that is of the incorrect type.

Some examples, given the following class to mock.

```
class MyThing:

    def convert_int_to_str(self, number: int) -> str:
        pass
```

And we try to specify an incorrect argument type to match against.

```
with tmock(MyThing) as my_mock:
    when(my_mock.convert_int_to_str("not an int")).then_return("hello")
```

We will get the following error:

```
typemock.api.MockTypeSafetyError: Method: convert_int_to_str Arg: number must be of␣
↪type:<class 'int'>
```

And if we try to specify the incorrect return type.

```
not_a_string = 3

with tmock(MyThing) as my_mock:
    when(my_mock.convert_int_to_str(1)).then_return(not_a_string)
```

We will get this error:

```
typemock.api.MockTypeSafetyError: Method: convert_int_to_str return must be of type:
↪<class 'str'>
```

And so, in summary, with typemock on strict mode and good type hints, it becomes difficult to make a mock that does something it should not do.

CHAPTER 4

Mocking an Object

The *DSL* for defining and using **typemock** takes much of its inspiration from the mocking libraries of statically typed languages, kotlin's mockk library in particular.

To mock a class or object, we use the *tmock* function. This returns a mock instance of the provided class type. This is actually a context manager, and you need to open the context to specify any behaviour.

You can pass a class or an instance of the class to the *tmock* function to be mocked. So…

```
with tmock(MyThing) as my_mock:

    # define mock behaviour
```

and:

```
with tmock(MyThing()) as my_mock:

    # define mock behaviour
```

are both acceptable.

There will be cases where, if an instance of the class has complex *__init__* functionality, then mocking a class will not be able to discover instance level attributes. In this case, you can attempt to mock an already initialised instance to resolve this. See more in the *Mocking Attributes* section.

Note:

- You must specify the behaviour of any method that your test is going to interact with. Interacting with a method with no specified behaviour results in an error.
- Typemock does not do static patching of the class being mocked. Any mocked behaviour will only be available from the mock instance itself, not via a class accessed call.
- Instance level attributes might not be available if the *__init__* method has some more complex logic. Use an already instantiated object in this case.

Now lets look at how to specify the behaviour for a mocked class or object.

4.1 Mocking Methods

First, let us define a class that we wish to mock.

```python
class MyThing:

    def return_a_str(self) -> str:
        pass

    def convert_int_to_str(self, number: int) -> str:
        pass

    def concat(self, prefix: str, number: int) -> str:
        pass

    def do_something_with_side_effects(self) -> None:
        pass
```

4.1.1 Simple response

```python
expected_result = "a string"

with tmock(MyThing) as my_thing_mock:
    when(my_thing_mock.return_a_str()).then_return(expected_result)

actual = my_thing_mock.return_a_str()

assert expected_result == actual
```

We also let the context of the mock close before we interacted with it, and it returned the response we had defined.

4.1.2 Different responses for different args

We can also specify different responses for different sets of method arguments as follows.

```python
result_1 = "first result"
result_2 = "second result"

with tmock(MyThing) as my_thing_mock:
    when(my_thing_mock.convert_int_to_str(1)).then_return(result_1)
    when(my_thing_mock.convert_int_to_str(2)).then_return(result_2)

assert result_1 == my_thing_mock.convert_int_to_str(1)
assert result_2 == my_thing_mock.convert_int_to_str(2)
```

4.1.3 Series of responses

We can specify a series of responses for successive calls to a method with the same matching args.

```python
responses = [
    "first result"
    "second result"
]

with tmock(MyThing) as my_thing_mock:
    when(my_thing_mock.convert_int_to_str(1)).then_return_many(responses)


for response in responses:
    assert response == my_thing_mock.convert_int_to_str(1)
```

By default, if we interact with the method more than the specified series, we will get an error. But you can set this to looping with the *loop* parameter for *then_return_many* responder.

4.1.4 Programmatic response

You can provide dynamic responses through a function handler.

The function should have the same signature as the method it is mocking so that mixes of positional and keyword arguments are handled in a deterministic way.

```python
def bounce_back_handler(number: int) -> str:
    return "{}".format(number)

with tmock(MyThing) as my_thing_mock:
    when(my_thing_mock.convert_int_to_str(1)).then_do(bounce_back_handler)

assert "1" == my_thing_mock.convert_int_to_str(1)
```

4.1.5 Error responses

We can also make our mock raise an Exception.

```python
with tmock(MyThing) as my_thing_mock:
    when(my_thing_mock.return_a_str()).then_raise(IOError)

my_thing_mock.return_a_str()  # <- Error raised here.
```

4.1.6 Arg Matching

Sometimes we want to be more general in the arguments needed to trigger a response. There is currently only the *match.anything()* matcher.

```python
with tmock(MyThing) as my_thing_mock:
    when(my_thing_mock.convert_int_to_str(match.anything())).then_return("hello")

assert "hello" == my_thing_mock.convert_int_to_str(1)
assert "hello" == my_thing_mock.convert_int_to_str(2)
```

Despite using this very broad matcher, any interactions with the mock will throw errors if they receive incorrectly typed args in their interactions.

4.1.7 Mocking async methods

We can also mock async methods. It just requires the addition an *await* key word when defining the behaviour. Here is an example:

```
#  Given some object with async methods.

class MyAsyncThing:

    async def get_an_async_result(self) -> str:
        pass

# We can setup and verify in an async test case.

async def my_test(self):
    expected = "Hello"

    with tmock(MyAsyncThing) as my_async_mock:
        when(await my_async_mock.get_an_async_result()).then_return(expected)

    assert expected == await my_async_mock.get_an_async_result()

    verify(my_async_mock).get_an_async_result()
```

Note: The the verify call does not need the *await* key word.

4.2 Mocking Attributes

Attributes are a little trickier than methods, given the layered namespaces of an instance of a class and the class itself.

With methods we can find the public members and their signatures regardless of if we are looking at an instance or a class. The state of a given instance/class implementation ie. its attributes can be defined in several ways, and so their type hints can be defined or deduced in several ways.

For now, *typemock* does its best to determine the type hints of attributes, and where it cannot, it is treated as untyped. Let's look at an example class to see what type hints are discoverable.

```
class MyThing:
    class_att = "foo"  # <- not typed
    class_att_with_type: int = 1  # <- typed, easy
    class_att_with_typed_init = "bar"  # <- type determined from __init__ annotation.
    class_att_with_untyped_init = "wam"  # <- not typed

    def __init__(
            self,
            class_att_with_typed_init: str,  # <- provides type for class level␣
↪attribute
            class_att_with_untyped_init,  # <- no type for class level attribute
            instance_att_typed_init: int,  # <- provides type for instance attribute
            instance_att_untyped_init,  # <- not typed
    ):
        self.class_att_with_typed_init = class_att_with_typed_init
        self.class_att_with_untyped_init = class_att_with_untyped_init  # <- not typed
        self.instance_att_typed_init = instance_att_typed_init  # <- type from init
```

(continues on next page)

(continued from previous page)

```
        self.instance_att_untyped_init = instance_att_untyped_init  # <- not typed
        self.instance_att_no_init: str = "hello"  # <- has a type hint, but not␣
↪discoverable = not typed
```

It might take some time to digest that, but essentially, effective attribute type hinting takes place either at a class level, or in the __*init*__ method signature.

If you pass in a class to the *tmock* function, typemock will try to instantiate an instance of the class so that it can discover instance level attributes. If some more complicated logic occurs in the __*init*__ method though, typemock may not be able to do this, and will log a warning. In this case, if you want to mock an instance level attribute you will need to provide an already instantiated instance to the *tmock* function.

To some up the basic guidelines for mocking attributes:

- Define your type hints at a class level or in the __*init*__ method signature.
- If the __*init*__ method of the class has some more complex logic, you may need to provide an instantiated instance to *tmock*

Depending on how this works in practice this may change, or some config may be introduced to assume attribute types from initial values.

With that quirkiness explained to some extent, let us look at how to actually mock an attribute. We will use this simpler class for the examples:

```
class MyThing:

        name: str = "anonymous"
```

Note: Currently, it is also not necessary to always specify behaviour of an attribute. It will by default return the value it was initialised with.

4.2.1 Simple Get

Just as with a method call, we can specify the response of a *get*.

```
with tmock(MyThing) as my_thing_mock:
    when(my_thing_mock.name).then_return("foo")

assert my_thing_mock.name == "foo"
```

4.2.2 Get Many

```
expected_results = [
   "foo",
   "bar"
]

with tmock(MyThing) as my_thing_mock:
    when(my_thing_mock.name).then_return_many(expected_results)

for expected in expected_results:
    assert my_thing_mock.name == expected
```

You can also provide the *loop=True* arg to make this behaviour loop through the list.

4.2.3 Get Raise

```
with tmock(MyThing) as my_thing_mock:
    when(my_thing_mock.name).then_raise(IOError)

my_thing_mock.name  # <- Error raised here.
```

4.2.4 Get programmatic response

As with methods, you can provide dynamic responses through a function handler.

It might be useful if you do want to wire up some stateful mocking/faking or have some other dependency.

```
def name_get_handler():
    return "my name"

with tmock(MyThing) as my_thing_mock:
    when(my_thing_mock.name).then_do(name_get_handler)

assert "my name" == my_thing_mock.name
```

CHAPTER 5

Verifying

Verification is important for checking that an interaction did or did not happen, or if it happened a specific amount of times. It can also allow for checking that interactions happened in a particular order.

Typemock currently only has quite limited verification, but it is good for most use cases.

5.1 Verifying Methods

Given the following class to mock:

```python
class MyThing:

    def convert_int_to_str(self, number: int) -> str:
        pass
```

We can verify method interactions in the following ways:

5.1.1 At least once

We can assert that an interaction happened at least once.

```python
with tmock(MyThing) as my_thing_mock:
    when(my_thing_mock.convert_int_to_str(match.anything())).then_return("something")

# Logic under test is called.

verify(my_thing_mock).convert_int_to_str(3)
```

5.1.2 Exactly

We can assert that an interaction happened a specific number of times.

```python
with tmock(MyThing) as my_thing_mock:
    when(my_thing_mock.convert_int_to_str(match.anything())).then_return("something")

# Logic under test is called.

verify(my_thing_mock, exactly=2).convert_int_to_str(3)
```

5.1.3 Never called

We can assert that an interaction never happened by checking for 0 calls.

```python
with tmock(MyThing) as my_thing_mock:
    when(my_thing_mock.convert_int_to_str(match.anything())).then_return("something")

# Logic under test is called.

verify(my_thing_mock, exactly=0).convert_int_to_str(3)
```

5.1.4 Any calls

And all of the previous examples have been verifying calls with specific args. We can also use the *match.anything* matcher to check for any interactions.

```python
with tmock(MyThing) as my_thing_mock:
    when(my_thing_mock.convert_int_to_str(match.anything())).then_return("something")

# Logic under test is called.

verify(my_thing_mock).convert_int_to_str(match.anything())
```

5.2 Verifying Attributes

Given the following class to mock:

```python
class MyThing:

        name: str = "anonymous"
```

We can verify interactions with its attribute as follows.

5.2.1 Get called at least once

```python
my_thing_mock =  tmock(MyThing)

# Logic under test is called.

verify(my_thing_mock).name
```

5.2.2 Get called exact amount of times

```
my_thing_mock =  tmock(MyThing)

# Logic under test is called.

verify(my_thing_mock, exactly=2).name
```

5.2.3 Set called with specific arg

```
my_thing_mock =  tmock(MyThing)

# Logic under test is called.

verify(my_thing_mock).name = 2
```

5.2.4 Set called with any arg

```
my_thing_mock =  tmock(MyThing)

# Logic under test is called.

verify(my_thing_mock).name = match.anything()
```

5.2.5 Set called exact amount of times

```
my_thing_mock =  tmock(MyThing)

# Logic under test is called.

verify(my_thing_mock, exactly=1).name = 2
```