Bright Yellow

by Ines Panker

WHO AM I

Hi, my name is **Ines Panker**. Software Engineer by profession, Explorer by mind.

Cheat Sheet Of Python Mock

1. Jun 2020 • Ines Panker

I always wanted to have this. The cool part of me, of course, wanted *me* to be the one who writes it, the pragmatic part just wanted to have access to a list like this and the hedonic part of me made me ignore the whole topic by telling me to chase after greater pleasures of life, at least greater than this blog post, no matter how magnificent it might eventually become, could ever be. But here I am, some years later, in the wrath of the epidemic lockdown, re-running Python tests in an infinite loop until I figure out which nobs and settings of this `mock` library I have to turn and set to get it to mock the damn remote calls.

The situation makes you wonder. Did it have to take a nationwide lockdown for me to start composing this list? Is it the dullness of the `mock` library that is the problem? Or is the culprit just a general aversion to spending any more time on tests than absolutely necessary? .. Who knows. But here we are, so let's get on with it.

`mock` is a great library. There are people out there, who are absolutely against `mock` and who will tell you that you should not mock anything ever. You should handle their stance just like you would the stance of any other absolutist: don't trust them. Writing tests is less about who has the moral upper ground and more about preventing bugs.

Level 1: Creating A New Mock Object

First, let's see what all the ways are we can create and configure a `Mock` object.

Attribute/Function we wish to mock	
Possible Mock objects	**Result of calling attr/func on Mock**
bool(book)	
book = Mock()	True
book = None	False
book.title	
book = Mock(title="Russian Literature")	"Russian Literature"
book = Mock(title="")	""
book = Mock()	<Mock name='mock.title' id='14039307'>
book.author.first_name	
book = Mock(author=Mock(first_name="Tatjana"))	"Tatjana"

book = Mock()
book.author.first_name = "Evgenij"
	"Evgenij"

book = Mock()	<Mock name='mock.author.first_name' id='140393072216336'>
book.author.get_full_name()	

book = Mock()
book.author.get_full_name.return_value = ""
	""

book = Mock(author=Mock(	

Attribute/Function we wish to mock	
get_fullname=Mock(return_value="Aleksandr Pushkin")))	"Aleksandr Pushkin"
Possible Mock objects	Result of calling attr/func on Mock

⚠ One special situation is the parameter `name`. The `Mock` class has a few input arguments, most of them (like `return_value`) are easy to remember. But then there is `name`, which nobody ever remembers. Add to this the fact that practically every class in the computer world has either a `title` or a `name` attribute and you have got yourself a perfect programmer trap.

author.name		

♡	author = Mock()
	author.name="Pushkin"
		"Pushkin" ♡

💀	author = Mock(name="Pushkin")	<Mock name='Pushkin.name' id='140504918800992'> 💀

Next to `Mock` there is also `MagicMock`. `MagicMock` can do all the same things that `Mock` can do, but it can do a smidgen more. `MagicMock` has some dunder (or magic) methods already defined: `__lt__`, `__gt__`, `__int__`, `__len__`, `__iter__`, ... This means we can do this: `len(MagicMock())` and receive `0`. Or this: `float(MagicMock())` and receive `1.0`. Or this: `MagicMock()[1]` and receive a new `MagicMock` object.

Implicitly Creating Sub-`Mock`s

When we are mocking a deeply nested attribute, we don't need to explicitly create sub-`Mock`s at every level. As soon as we access an attribute/function/property, a new `Mock` will automatically be created, if none exists. And because calling the same attribute will return the same sub-`Mock` every time, we can use this to our advantage and simply say `mock_object.author.country().title = "Lichtenstein"` instead of `mock_object.author = Mock(country=Mock(return_value=...))`. But the rule of thumb is that the path must consist solely of functions and attributes. The `MagicMock` can handle a few more things in the path, like `[0]`.

Examples of paths thatare ok:

- `book.get_review(sort="date", order="desc").reviewer.get_country().short_name`
- `book()()()`

As soon as a non-function comes along, we get an `Exception`. These paths are not ok:

- `book.reviews[0]`
- `len(book.reviews)`
- `book.year > 1950`

book.data["reviews"][0].reviewer.date	

book = **Mock**(
 data={
  "reviews":[
   Mock(reviewer=Mock(
    date=datetime.now()
   ))
  ]
 }
)
	datetime.datetime(2020, 5, 17, ...)

book = Mock()	TypeError: 'Mock' object is not subscriptable *The problem is with the '["reviews"]'-part.*
book = **MagicMock()**	<MagicMock name='mock.data.__getitem__().__getitem__().reviewer.date' id='14050491'>
book = **MagicMock()** book.data["reviews"][0].reviewer.date = datetime.now()	datetime.datetime(2020, 5, 17, ...)

len(book.data["reviews"])

book = MagicMock() book.data["reviews"].__len__.return_value = 120	120

book.get_review(type="oldest").reviewer.get("name", "unknown")

book = Mock()
book.get_review.return_value.reviewer = {"name": "Natalia"}
	"Natalia"

book = Mock()
book.get_review.return_value.reviewer = {}
	"unknown"

book = Mock(
  **{"get_review.return_value.reviewer": {}}
)
	"unknown"

book.all_reviews[-1].get_country(locale="en").reviewer.get("name")

book = MagicMock()
book.all_reviews[-1].get_country(locale="en").reviewer = {"name": "Natalia"}
	"Natalia"

If you don't know the exact path you have to take to get to the value you wish to mock, just create a `Mock()` object, call the attribute/function and print out the mocked value, that is produced.

Let's say I have: `a.b(c, d).e.f.g().h()`. When I create `a = Mock()` and call the previous code I receive this `Mock` object: `<Mock name='mock.b().e.f.g().h()' id='14050'>`. This tells me the exact path I have to mock. I copy the path and just replace every `()` with `.return_value` and get: `a.b.return_value.e.f.g.return_value.h.return_value = "My Result"`.

Level 2: Missing Attributes

item.slug -> should raise AttributeError

book = Mock(**spec=[]**)	AttributeError: Mock object has no attribute 'slug'

getattr(item, "slug", None) or getattr(item, "uid", None)

item = Mock(uid="12345", **spec=["uid"]**) *Making sure there is no 'slug' attribute.*	"12345"

What if we have a common utility function, which extracts the correct title from any object. It might look like this:

```
def extract_title(item: Union[Book, Author, Review]) -> str:
  if isinstance(item, Book):
    return item.title

  if hasattr(item, "get_full_title"):
    return item.get_full_title()

  if hasattr(item, "summary"):
    return item.summary

  raise Exception("Don't know how to get title")

# ---- models: ---------------

@dataclass
class Book:
  title: str

class Author:
  def get_full_title() -> str:
    pass

class Review:
```

```
@property
def summary() -> str:
  pass
```

To test `extract_title` with objects mocking all 3 classes (`Book`, `Author`, `Review`), we can resort to `Mock`'s `spec` attribute. `spec` can be a list of attributes as shown above or it can be a **class** or a class **instance** or anything really as long as `Mock` can call `dir(spec_obj)` on it.

Calling `Mock(spec=Author)` is the same as: `Mock(spec=[attr for attr in dir(Author) if not attr.startswith("__")]`. If we try to access any attribute not on the spec-list, an `AttributeError` is raised.

extract_title(item)	
default_mock_obj = Mock()	Will alwas return True for `hasattr(item, "get_full_title")`: <Mock name='mock.get_full_title()' id='1405'>
mock_book = Mock(**spec=Book("")**)	<Mock name='mock.title' id='140504'>
mock_book = Mock(spec=Book(""), title="Russian Literature")	"Russian Literature"
mock_author = Mock(**spec=Author**)	<Mock name='mock.get_full_title()' id='14050'>
mock_book = Mock(spec=Author) mock_author.get_full_title.return_value = "Aleksandr Pushkin"	"Aleksandr Pushkin"
mock_review = Mock(**spec=Review**)	<Mock name='mock.summary' id='14050'>
mock_review = Mock(spec=Review) mock_review.summary = "Oh, bitter destiny"	"Oh, bitter destiny"

Level 3: The Attribute With The Very Unfortunate Name - `Side_effect`

Every `Mock` object can be instantiated with a `side_effect` attribute. This attribute is useful in 3 types of situations and only one of them can honestly be called a side effect. Unfortunately, the lamentable name makes developers miss how versatile and convenient its underlying functionality actually is.

`Side_effect` As An Iterator

We have a remote call to Slack where we fetch all channels of some Slack workspace. But Slack's API is in its Nth version and has thus the annoying, but very sensible, limitation built-in which permits us to only fetch 100 channels at a time. To get the 101st channel we have to make a new GET request with the pagination cursor we received in the last request. Here is the code:

```
def fetch_one_page_of_slack_channels(cursor=None):
  ....<channels: List[str]>....
  return channels, next_page_cursor

def fetch_all_slack_channels():
  all_channels: List[str] = []
  fetched_all_pages = False
  cursor = None
  while not fetched_all_pages:
    new_channels, cursor = fetch_one_page_of_slack_channels(cursor)
    all_channels.extend(new_channels)
    fetched_all_pages = bool(cursor is None)

  return all_channels
```

Now we want a test making sure that `fetched_all_pages` doesn't stop after the first page.

fetch_all_slack_channels()	
fetch_one_page_of_slack_channels = Mock()	TypeError: cannot unpack non-iterable Mock object

Explanation: `Mock` will return a new `Mock` object for every call, but 1 `Mock` cannot be unpacked into 2 items: `new_channels` and `cursor`:
`----> new_channels, cursor = fetch_one_page_of_slack_channels(cursor)`

fetch_one_page_of_slack_channels = Mock(side_effect=**[** (["channel_1","channel_2"], "__NEXT__PAGE__"), (["channel_3"], None), **]**)	['channel_1', 'channel_2', 'channel_3']

`Side_effect` As An Exception

This one is for testing Exception-handling code.

Let's say you are letting your users log in with any social account they choose. But how do you handle the event where 2 of your users want to connect the same GitHub account? We want to catch this event and transform it into a friendly error message for the user.

Here is the code:

```
def connect_github_account(user, github_uid):
  try:
    user.social_accounts.add("github", github_uid)
  except SocialAlreadyClaimedException as exc:
    logger.log('...', exc_info=exc)
    return False, "Sorry, we could not connect you"
  ...
```

How do we test for this?

user.social_accounts.add("github", github_uid) -> Must raise SocialAlreadyClaimedException	

user = Mock()
user.social_accounts.add.side_effect = SocialAlreadyClaimedException
	(False, 'Sorry, we could not connect you')

user = Mock()
user.social_accounts.add.side_effect = SocialAlreadyClaimedException(
"GitHub account XXX already connected to user AAA"
)
	(False, 'Sorry, we could not connect you')

The return value for the function is the same. The difference is in the message of the exception. In the first example, the exception is raised without any message and in the second our message is supplied. We could check this by mocking the `Logger` and looking at the calls that were made to it: `logger.call_args` -> `call('...', exc_info=SocialAlreadyClaimedException('GitHub account XXX already connected to user AAA'))`

`Side_effect` As A Substitute Class/Function (Not A Mock Object)

The last and most awesome use for `side_effect` is to use it to replace the contents of functions or classes.You can specify a new function/class which will be used instead of the original function/class. But it must be a function or a class not a different type of object and it must accept the same variables as the original function/class.

For example, we want to write a unit test for our menu. The links that we show on the menu depend on who is logged in and what organization they belong to. The code looks like this:

```
def menu_urls(user):
  org_settings_url = create_url(endpoint="org_settings", org_slug=user.org.slug)
  dashboard_url = create_url(endpoint="dashboard")
  user_settings_url = create_url(endpoint="user_settings", org_slug=user.org.slug, user_slug=user.slug)
  ...

  return [org_settings_url, dashboard_url, user_settings_url, ... ]
```

We don't want to set up the whole app for this test, we want to write the simplest unit test, just making sure that the links are created with the correct arguments.

One solution is to mock `create_url` with the `side_effect` option.

menu_urls(user)

def **substitue_create_url**(endpoint, **kwargs):
  return f"{endpoint} WITH {kwargs}"
 	
create_url = Mock(side_effect=substitue_create_url)
 	
user = Mock(**{
  "slug": "__USER__SLUG__",
  "org.slug": "__ORG__SLUG__"
})
	[
	  "org_settings WITH {'org_slug': '__ORG__SLUG__'}",
	  'dashboard WITH {}',
	  "user_settings WITH {'org_slug': '__ORG__SLUG__', 'user_slug': '__USER__SLUG__'}"
	]

Each time that `create_url` is called inside `menu_urls`, what is actually called is our `substitue_create_url`. It is of course called with the same input arguments, which we can modify or store in any way.

def **substitue_create_url**(**kwargs):
  return "_SOME_URL_"
 	
create_url = Mock(**side_effect=substitue_create_url**)
 	
user = Mock()
	['_SOME_URL_', '_SOME_URL_', '_SOME_URL_']

This is the same as if we had set up `create_url = Mock(return_value="_SOME__URL_")`.

The same thing can be done with classes. Let's say I store some user configuration in a class and I build it up step by step. The code looks like this:

```
def create_config(user):
  config = Configuration()
  if user.can_read():
    config.set("literate", True)
  if user.can_jump():
    config.set("springy", True)
  ...
  return config
```

One possible way to write a test for this would be to mock the `Configuration` class.

create_config(user_1)
create_config(user_2)

```
class SubstituteConfiguration:
  def __init__(self):
    self.config = {}

  def set(self, key, value):
    self.config[key] = value

  def __repr__(self):
    return str(self.config)

Configuration = Mock(side_effect=SubstituteConfiguration)

user_1 = Mock(**{
  "can_read.return_value": False,
})
user_2 = Mock()
```

```
{'springy': True}
{'literate': True, 'springy': True}
```

When using `side_effect` in this way, be careful. Don’t go overboard with mocking. These tests can be very unstable. Even non-functional changes to the source like replacing a positional argument with a keyword argument can make such a test fail.

`Side_effect` Vs `Return_value`

> With `side_effect` you define a function/class (or iterator or exception), which should be called instead of the original function/class. With `return_value` you define the result, what the function/class is supposed to return so that we don’t need to call it.

Level 4: Tracking Calls

Every `Mock` remembers all the ways it was called. Sometimes we want to control what a mocked function/class returns, other times we want to inspect how it was called.

`Mock` objects come with built-in assert functions. The simplest one is probably `mock_obj.assert_called()` and the most commonly used might be `mock_obj.assert_called_once_with(...)`. With the help of `assert`-functions and only occasionally by inspecting the attributes `mock_obj.call_args_list` and `mock_call_args` we can write tests verifying how our objects are accessed.

All useful functions:
mock_obj.assert_called()
mock_obj.assert_called_once()
mock_obj.assert_called_with(100, "Natalia")
mock_obj.assert_called_once_with(100, "Natalia")

from mock import **ANY**
mock_obj.assert_called_once_with(**ANY**, "Natalia")
When we don't care to know all function parameters or don't care to set them all, we can use ANY as a placeholder.

When we don't care to know all function parameters or don't care to set them all, we can use ANY as a placeholder.

A call on a Mock object	**Assert calls, which will not raise an error**
mock_obj(100, "Natalia")	mock_obj.assert_called()
	mock_obj.assert_called_once()
	mock_obj.assert_called_with(100, "Natalia")
	mock_obj.assert_called_once_with(100, "Natalia")

♡	from mock import **ANY**
	mock_obj.assert_called_once_with(**ANY**, "Natalia")
	When we don't care to know all function parameters or don't care to set them all, we can use ANY as a placeholder.

What about when the mocked function is called more than once:

mock_obj(100, "Natalia")
mock_obj(None, "Evgenij")

mock_obj.assert_called()
mock_obj.assert_called_with(None, "Evgenij") *The assert_called_with compares only to the last call. Had we checked for the "Natalia"-call it would raise an error.*
mock_obj.assert_any_call(100, "Natalia")

from mock import **call**
mock_obj.assert_has_calls([
  call(None, "Evgenij"),
  call(100, "Natalia"),
], any_order=True)

For when we want to make sure that something didn't happen we can use `assert_not_called()`. Example: we send Slack messages to those users, who have opted-in to receiving Slack messages and not to others.

```
def send_slack_msg(user):
  if user.has_slack_disabled():
    return

  slack_post(...)
```

user=Mock(**{"has_slack_disabled.return_value": True})
slack_post = Mock()
send_slack_msg(user)

slack_post.assert_not_called()

Should we want to make more calls to the same mock function, we can call `reset_mock` in between calls.

send_slack_msg(user_1)
send_slack_msg(user_2)

slack_post.assert_called_once_with(...)
slack_post.**reset_mock()**
slack_post_assert_called_once_with(...)

Level 5: Mocking Imports With `Patch`

The last piece in our puzzle of mocking is `patch`. Until now we've pretended that we have complete access to all functions and variables we are testing, that the test and the source code live in the same context, the same file, but this is never true. So, if the source file *sending.py* looks like this:

```
#  sending.py
def send_post(...):
  ...

def send_slack_msg(user):
  if user.has_slack_disabled():
    return

  slack_post(...)
```

and we want to write a test in the file `test_sending.py`, how can we mock the function `send_post`?

By using `patch` .. and giving it the full path to the *relative* location/target of the thing we want to mock.

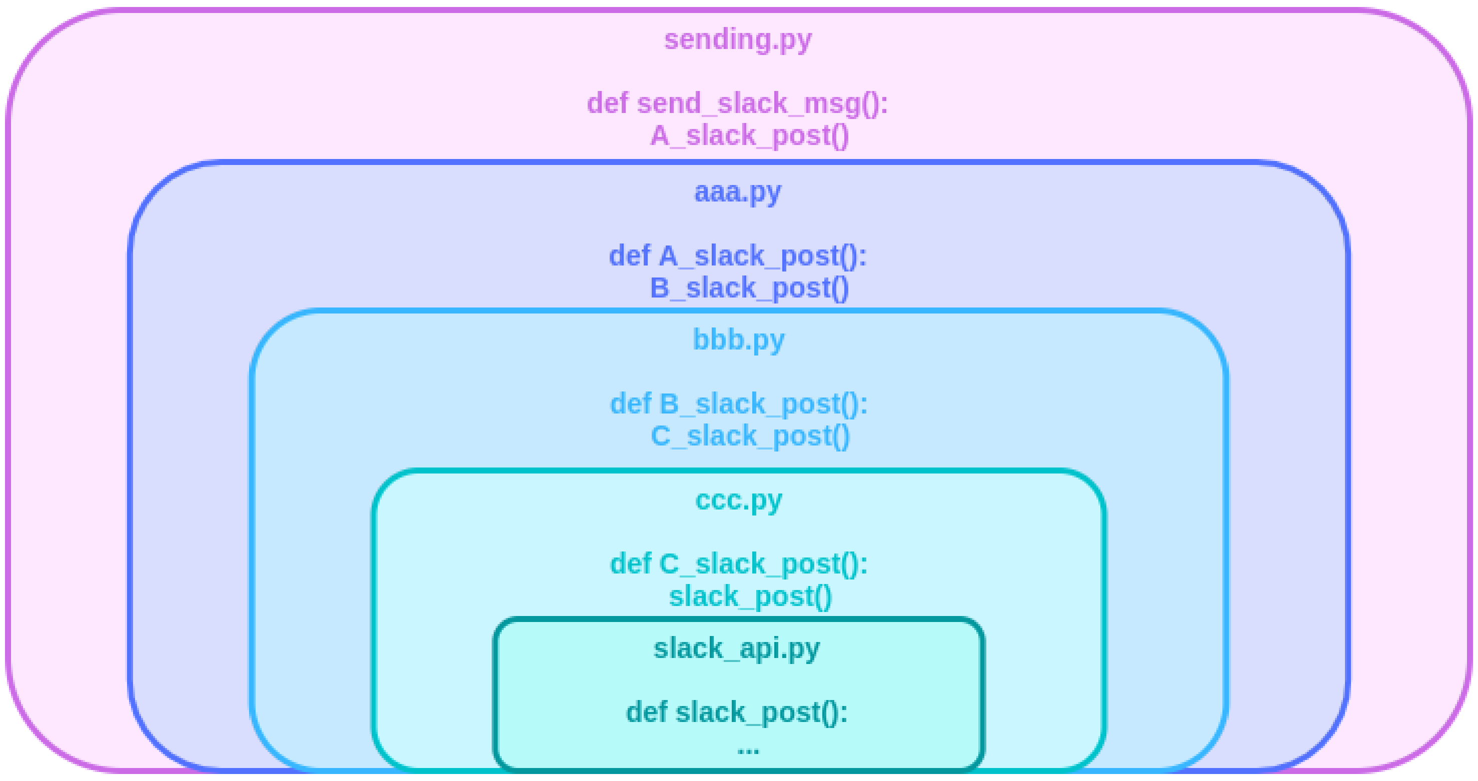
> Generally speaking, the **target** is constructed like this: **<prefix><suffix><optional suffix>**.
> *The prefix is*: the path to the module, which will import the function/class/module we are mocking.
> *The suffix is*: the last part of the `from .. import..` statement which imports the object we are mocking, everything after `import`.
> *The optional suffix is*: If the `suffix` is the name of a module or class, then the optional suffix can the a class in this module or a function in this class. This way we can mock only 1 function in a class or 1 class in a module.

`patch` can be used as a decorator for a function, a decorator for a class or a context manager. In each case, it produces a `MagicMock` (exception: `AsyncMock`) variable, which it passes either to the function it mocks, to all functions of the class it mocks or to the with statement when it is a context manager.

Patch Target - Examples Of Prefix-Suffix-Optional_suffix Combinations

Here is a layered series of calls: `slack/sending.py` calls `aaa.py`, which calls `bbb.py`, which calls `ccc.py`, which calls `slack_api.py`.

In a diagram:



and in code:

```
#  slack/sending.py
from a_lib.aaa import A_slack_post

def send_slack_msg(user):
  print(f"send_slack_msg will call {A_slack_post}")
  A_slack_post(user)


#  a_lib/aaa.py
from b_lib.bbb import B_slack_post

def A_slack_post(user):
    print(f"A will call {B_slack_post}")
    B_slack_post(user)


# b_lib/bbb.py
from c_lib.ccc import C_slack_post

def B_slack_post(user):
    print(f"B will call {C_slack_post}")
    C_slack_post(user)


# c_lib/ccc.py
from slack_lib.slack_api import slack_post


def C_slack_post(user):
    print(f"C will call {slack_post}")
    slack_post(user)


# slack_lib/slack_api.py
def slack_post(user):
  print(f"send Slack msg")
```

What is the patch `target` if we want to only mock what is inside `slack_post`? What if we want to mock everything from `B_slack_post` on?

we will call `send_slack_msg(user)`and mock `slack_lib.slack_api.slack_post`

@patch("**c_lib.ccc.**slack_post")
def test(mock_slack_post):
  send_slack_msg(...)
  mock_slack_post.assert_called()

Prefix: the path to the file, where we want to start mocking = `c_lib/ccc.py`*.*
Suffix: name of our function = `slack_post`*.*

Our printout will be:
send_slack_msg will call <function A_slack_post at 0x7f25>
A will call <function B_slack_post at 0x7f25>
B will call <function C_slack_post at 0x7f25>
C will call <MagicMock name='slack_post' id='1397'>

we will call `send_slack_msg(user)`and mock `b_lib.bbb.B_slack_post`

@patch("**a_lib.aaa.**B_slack_post")
def test(mock_B_slack_post):
  mock_B_slack_post.return_value = True

send_slack_msg(...)

Prefix: the path to the file, where we want to start mocking = `a_lib/aaa.py`.
Suffix: name of our function = `B_slack_post`.

Our printout will be:
send_slack_msg will call <function A_slack_post at 0x7fc>
A will call <MagicMock name='B_slack_post' id='1405'>

For reference, when nothing is mocked and we call `send_slack_msg(user)`, the following is printed:

send_slack_msg will call <function A_slack_post at 0x7f25>
A will call <function B_slack_post at 0x7f25>
B will call <function C_slack_post at 0x7f25>
C will call <function send_slack_msg at 0x7f25>
send Slack msg

Patching Only One Function In A Class

```
#  slack_lib/slack_api.py
from slack import slack_api

def send_slack_msg(user):
  slack_instance = slack_api.SlackAPI(user)
  print(f"send_slack_msg will call {slack_instance.post} on {slack_instance}")
  slack_instance.post()


#  slack_lib/slack_api.py
class SlackAPI:
  def __init__(self, user):
    self.user = user

  def post(self):
    print(f"send from class to user: {self.user}")
```

we will call `send_slack_msg` and mock only `SlackAPI.post`:

@patch("slack.sending.**slack_api.SlackAPI.post**") def test(mock_slack_api_post):
mock_slack_api_post = lambda :"Mock was called"
send_slack_msg(Mock())

Prefix: path to module where `send_slack_msg` lives = slack.sending.
Suffix: what is after `import` in `from slack import slack_api` = slack_api.
Optional suffix: a class inside `slack_api` + a function inside the class = SlackAPI.post

Our printout will be:
send_slack_msg will call <MagicMock name='post'> on <slack_lib.slack_api.SlackAPI>
Mock was called

Patch `Object`

As far as I can see `patch.object()` works exactly the same as `patch()`. The only difference is that `patch` takes a string as the target while `patch.object` needs a reference. `patch.object` is thus used for patching individual functions of a class.

For the above situation the pather would look like this:

we will call `send_slack_msg` and mock only `SlackAPI.post`:

from slack_lib.slack_api import SlackAPI
@patch.object**(SlackAPI, "post")**
def test(mock_slack_post):

Knicks-Knacks A: `PropertyMock`

For when you need to mock a `@property`:

```
# models/user.py
class User:
  @property
  def name(self):
    return f"{self.first_name} {self.last_name}"
  @name.setter
  def name(self, value):
    self.first_name = value

# service/user.py
def create_user(name):
  user = User()
  user.name = name
  return user.name
```

mock the property `name`:

from mock import PropertyMock
from models.user import User
from services.user import create_user

@patch.object(User, "name", new_callable=PropertyMock)
def test(mock_user):
  mock_user.return_value = "Jane Doe"
  assert create_user("Janette") == "Jane Doe"
  mock_user.assert_any_call("Janette")

Whenever we will call user.name, we will get the string "Jane Doe". But the mock also remembers all the values it was called with.

Knicks-Knacks B: Patching A Constant

How do we mock a constant? With `patch`'s `new`-attribute:

```
# constants.py
MSG_LIMIT = 7

# code.py
from constants import MSG_LIMIT

def get_latest_msgs():
  return messages.limit(MSG_LIMIT).all()
```

mock `MSG_LIMIT` and set it to 3

@patch("code.MSG_LIMIT", new=3)

Knicks-Knacks C: Having Many `Patch`-Ers

Each patcher can pass a `MagicMock` variable to the calling function. Because this is Python, the decorators are applied bottom up:

```
@patch("path_1")
```

```
@patch("path_2")
@patch("path_3")
def test(mock_path_3, mock_path_2, mock_path_1):
  ...
```

More!

This was supposed to be a list of essential mock functionalities. It covers 99% of everything I ever needed or saw mocked. Certainly, this is not the limit of the `mock` library, but I'm already looking forward to utilizing this summarized version of how `Mock`s should be constructed instead of reading through the longer (and more precise) official documentation or googling various StackOverflow answers. And who knows, maybe it will someday be of use to some other `mock`-frustrated programmer soul, who will be gladdened to have found this post.

External Sources

- Python docs: mock object library

Found a typo or a mistake? Edit this page