Mocking

The Mock Turtle: Illustration by Sir John Tenniel, from Lewis Caroll's famous novel Alice in Wonderland. The Mock Turtle "...is a very melancholy character that pines pathetically for the days when it was once a real turtle." (Alice in Wonderland fandom Wiki)[1]
Hopefully the mocks that you create will not become self-aware.

As we've discussed in the week 9 notes, you will often need to create code that simulates the behaviour of "real" parts of the system that AREN'T the primary target of the test, for testing purposes.

These test doubles may be required because:

- the real code that you need to bypass (we'll call this the "target for doubling") has unpredictable behaviour - for instance, it interacts with an external system or users.
- the target for doubling doesn't actually exist yet.

[1] http://aliceinwonderland.wikia.com/wiki/The_Mock_Turtle

- the target for doubling is slow, making the unit tests run slowly. Remember, you want to run unit tests all the time.
- the target for doubling interacts with software or hardware that doesn't exist in the unit testing environment.
- you want to test your test target with cases that are rarely if ever generated by the target for doubling, such as error cases.
- you want to inspect how the code you're actually testing calls the target for doubling, and the target doesn't provide facilities for doing this.

In these notes, we'll discuss how to use Python's built-in mock library for achieving this.

Handcrafted test doubles.

But first, an observation. In statically-typed languages like Java, it's rather a pain to write test doubles and execute with them; you end up defining lots of interfaces or abstract base classes, one implementation contains the real code, and you have one (or more) faked implementations for testing purposes.

In a dynamically typed language like Python, by contrast, it's often quite easy to make a test double.

Consider the following class for counting the number of occurrences of each character in a file:

```
class CharCounter:
    '''Class for counting the number of instances of characters in a
file'''
    def __init__(self, infile):
        """Construct the class

        Arguments:

        infile: the file object to count
        """
        self.counts = {}
        self.buildcounts(infile)

    def buildcounts(self, infile):
        """Build the count for a file.  This is called by the constructor
        so should not typically be called directly
        """
        chars = infile.read()

        for char in chars:
            if char in self.counts:
                self.counts[char] += 1
            else:
                self.counts[char] = 1

    def getcount(self, letter):
        """Look up the count for a particular character

        Arguments:

        letter: the character to get the count for

        Returns: the number of times the letter occurs, which may be
        zero if it is not present
        """
        if letter in self.counts:
            return self.counts[letter]
        else:
            return 0
```

One case where we will commonly want to use test doubles rather than real code is when the real code performs a relatively slow I/O operation, such as accessing a disk (or, especially, a network).

Digression: Disks (even SSDs) are slow, Networks are glacial

While your introductory algorithms and data structures class doesn't really teach you this, in many applications you will write, by far the slowest part of the system is transferring stuff into and out of memory via disk or networks. Even the fastest solid-state hard disks are 50-100x slower than RAM, particularly when writing data. Retrieving things from a network, particularly if the server is located on another continent, is even slower; the round-trip time to request a single piece of information with a client located in Australia and a server in the United States is often over a second. Imagine if you've got hundreds of tests, each making multiple such requests!

One of the key advantages of good unit tests is that you can run them very regularly - basically, every time you get the code to a runnable state, you can run the tests to see if the code works. If your tests take half an hour to run, that's not practical. Therefore, if your tests are slow because of disk or network accesses, it's a very good idea to use mocking to speed them up.

Getting back to our example, it's very easy to use hand-written test doubles to do so here. Python code typically takes the "duck typing" approach - it cares whether an object walks and quacks like a duck, rather than whether it's part of the Duck type hierarchy. Specifically, here, to test the `CharCounter` class, we need an object that implements a read method that returns a list (or other iterable) of single-character strings.

We can make one of those easily enough without delving into the `mock` module, or any advanced Python magic:

```
class FakeFile:
    def __init__(self, contents):
        self.contents = list(contents)

    def read(self):
        return self.contents
```

We can then use a `FakeFile` object as a test double (a stub) to test the `CharCounter` without actually accessing files:

```
from charcounter import CharCounter
from fakefile import FakeFile
import unittest


class CharCounterHandTest(unittest.TestCase):
    def test_one(self):
        infile = FakeFile("fredfredfred")
        counter = CharCounter(infile)
        self.assertEqual(counter.getcount("f"), 3)

if __name__=="__main__":
    unittest.main()
```

You might therefore be wondering what the point of a special library for making test doubles is.

Problem #1: intercepting indirect function/method invocations

Consider this modified version of `CharCounter`, where, rather than passing a file object as a parameter, you pass in a file path as a string:

```
class CharCounter:
    '''Class for counting the number of instances of characters in a
file'''
    def __init__(self, inpath):
        """Construct the class

        Arguments:

        inpath: a path to the fil to count
        """
        self.counts = {}
        self.buildcounts(inpath)

    def buildcounts(self, inpath):
        """Build the count for a file.  This is called by the constructor
        so should not typically be called directly
        """
        infile = open(inpath, "r")
        chars = infile.read()
```

```
        for char in chars:
            if char in self.counts:
                self.counts[char] += 1
            else:
                self.counts[char] = 1
 # Rest of the code is unchanged....
```

We can't simply pass in a `FakeFile` object. To test our alternate version of `CharCounter` without accessing files, we need to somehow trap calls to `open()` and get it to return a test double for `infile`.

In a statically typed language like Java, replacing the real `open()` with a fake `open()` would be extremely difficult. It's actually not as hard in Python, as this example (non-test) code demonstrates:

```
from fakefile import FakeFile

def myopen(fakefile, fakemode):
    return FakeFile("this is a test")

properopen = open

#substitute in the fake open function
open = myopen

#calls the fake open function and prints "this is a test"
myfake = open("/proc/version_signature","r")

print(myfake.read())

#switch the real open back in
open = properopen

myreal = open("/proc/version_signature", "r")
print(myreal.read())
```

If you don't understand what this code is doing, spend some time playing with it in a Python interpreter until you do.

Now, the fact that Python has first-class functions[2] and lets you do this kind of thing without horrible low-level hacks is nice, but doing this kind of thing manually has (at least) a couple of problems:

- You have to be very disciplined about the scope of your substitution, as there could be all manner of bugs. This is quite complicated in unit testing, as the test runners may execute your tests in *any* order.
- The code for substituting one function for another was relatively simple. Substituting an instance method attached to a class results in considerably more complex code.

Wouldn't it be nice if there was one simple way to do this kind of substitution?

Problem #2: monitoring the way functions are invoked

The point of unit testing is to check whether the software under test actually behaves as expected. Those expectations can include that it invokes other code correctly.

To take a very simple example, we might want to check that `CharCounter` is opening the file we ask it to count, and that the read mode is "r".

Again, it is *possible* to handcraft test doubles to do this (if you want to learn about some fairly advanced aspects of Python, it's a good exercise to attempt to do it cleanly), but the code to do so gets increasingly complex as the checking becomes more elaborate.

Problem #3: a maze of twisty little attributes, all alike[3]

Finally, there's also simple tedium in that you have to create doubles for every single attribute (or at least, every single attribute invoked by the test), even if you don't ever use the returned values. That gets very tedious, very quickly!

The solution - the `mock` library

`mock` is a library distributed with all recent versions of Python that helps with all three of these issues.

`mock` lets you:

---

[2] https://mitpress.mit.edu/sicp/full-text/book/book-Z-H-12.html#call_footnote_Temp_121
[3] http://rickadams.org/adventure/

- Easily create test doubles that provide sensible behaviour for attributes and methods.
- Substitute ("patch") indirectly called functions and newly created objects with mocks.
- Check the way that mocked objects are accessed during tests to see whether your code is behaving as expected.

While it's generally (not always, but generally) much easier to use `mock` than hand-create test doubles, that doesn't mean it's easy. You're essentially rewiring your program for testing, selectively bypassing many of the layers of abstraction that Python provides for you. So it's not surprising that it can be challenging to use.

Generally, digging down in the guts of your program to substitute one method call for another is something you should only do during testing and debugging (unless you're a devotee of aspect-oriented programming, which advocates doing this kind of thing for fun and profit[4]). However, to demonstrate how the mock library works, we'll initially use examples that are *not* in the context of a unit test.

Mock objects

The point of the `mock` library is to replace real objects with mock objects. The basic functionality of this kind of mocking comes from the `Mock` class.

Just for demonstration purposes, we'll create an instance of `Mock` in an interactive Python session and show some of what it can do:

```
xubuntu@xubuntu-VirtualBox:~/Desktop$ python3
Python 3.5.2 (default, Nov 23 2017, 16:37:01)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from unittest.mock import Mock
>>> x=Mock()
>>> print(x)
<Mock id='140350892893632'>
>>> print(x.y)
<Mock name='mock.y' id='140350892893576'>
>>> print(x("some", "random", "arguments", 12345))
<Mock name='mock()' id='140350892948224'>
>>> print(x.z())
<Mock name='mock.z()' id='140350892893688'>
>>> x.y.return_value=1
>>> print(x.y())
```

[4] https://msdn.microsoft.com/en-us/magazine/dn574804.aspx

```
1
>>> x.y.assert_called_once()
>>> x.y.assert_not_called()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.5/dist-packages/mock/mock.py", line 897, in
assert_not_called
    raise AssertionError(msg)
AssertionError: Expected 'y' to not have been called. Called 1 times.
>>> print(x.y())
1
>>> x.y.assert_called_once()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.5/dist-packages/mock/mock.py", line 915, in
assert_called_once
    raise AssertionError(msg)
AssertionError: Expected 'y' to have been called once. Called 2 times.
```

There's a lot to digest here, so let's take it step by step.

Firstly, creating a mock object is straightforward - we can just import the `Mock` class from the `mock` library, and create one as we would any other object.

The unique stuff starts happening in the third line of user input: `print(x.y)`. By default, when *any* reference to a non-existent attribute of a `Mock` object occurs, the `Mock` library magically replaces that with...another instance of a `Mock` object. This occurs *recursively*. We haven't shown it, but if we tried to reference `x.y.fred`, that would give you back another `Mock` object, too.

`Mock` objects are *callable* - that is, you can treat them as a function and call them. By default, calling a `Mock` object, regardless of the arguments passed to it in the call, returns yet another `Mock` object!

Sometimes, you'll need more than that - you'll need to control what the mock returns. You can do this (while retaining all the other nice properties of mocks) by setting the mock's `return_value` attribute. If you need more sophisticated behavior than this, you can use the `side_effect` attribute, which lets you retrieve values from a list or other iterable object if the mock is called repeatedly. It can also just invoke a function that returns a value, and as the name implies, can have any side effects you want. See the official documentation if you need to use `side_effect`.

Finally, we demonstrate the ability to check that the object we've mocked is being used correctly. `Mock` provides a range of assertions to fully check the arguments a mock is being called with, but we'll use some simple ones to check how many times the mock has been called.

When the assertion is true, the assertion succeeds silently; when it's false, an AssertionError is raised. So in this example we do the following:

- call `x.y()` once
- assert that `x.y()` has been called once, which is `True`
- then assert that it hasn't been called, which causes an `AssertionError` to be raised.
- We then call `x.y()` again, and assert that `x.y()` has been called once, which is no longer true (it's been called twice) and therefore causes an assertion error to be raised.

Children of Mock

No, not that kind of mocking child. (Photo: Stewie Griffin plays Mozart, still from "Family Guy". Copyright held by the owners)[5]

You can use plain `Mocks` in your tests without any further ado, but the `mock` library has a couple of very useful subclasses of `Mock`. Consider if you want to mock a list-like object on which the `len()` function is called. `Mocks`, by default, have no implementation of `len()`.

[5] https://youtu.be/CgjBqPIPbVY?t=36s

```
Python 3.5.2 (default, Nov 23 2017, 16:37:01)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from mock import Mock
>>> listlike = Mock()
>>> len(listlike)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'Mock' has no len()
>>>
```

The `MagicMock` class, by contrast, implements many of Python's "magic" functions, including `len()`, and returns allegedly sensible default values:

```
xubuntu@xubuntu-VirtualBox:~/gdrive/teaching/teaching resources/FIT2107
Lecture Notes/mocking_code$ python3
Python 3.5.2 (default, Nov 23 2017, 16:37:01)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from mock import MagicMock
>>> listlike = MagicMock()
>>> len(listlike)
0
>>>
```

The real value of `MagicMock` is that you can use `return_value` to set return values for the magic methods in one simple assignment:

```
>>> from mock import MagicMock
>>> listlike = MagicMock()
>>> len(listlike)
0
>>> listlike.__len__.return_value=10
>>> len(listlike)
10
>>>
```

`MagicMock`'s extra functionality comes at no cost, so, as the official documentation[6] puts it :

> In most of these examples (in the official docs) the `Mock` and `MagicMock` classes are interchangeable. As the `MagicMock` is the more capable class it makes a sensible one to use by default.

There are several more children of `Mock` available from the library. `NonCallableMock` and `NonCallableMagicMock`, like the name says, can't be called as a function - if you don't want that functionality, it's a good idea to use them. We'll come back to this later. Finally, there's also a `PropertyMock` specifically for mocking Python properties. If you need this functionality, read the official documentation.

A simple example

Now, let's take what we've learned about mocking to see if we can use it for unit testing.
One of the most common situations where mocking is useful is for network accesses. Gitlab, as most of you should be aware by now, is a version control system which makes it easy to create, manage, and monitor git repositories and other useful project artifacts like task boards, wikis, and other material. As well as its user-facing web interface, the Gitlab software can be accessed through a REST API, which makes it easy to write software that interacts with Gitlab. `python-gitlab` is a library which simplifies this interaction.

We're going to write some utility functions that use the `python-gitlab` library to extract useful information out of the gitlab API. Specifically, let's write one to extract the name of the first repository owned by the user that matches a keyword search:

```
import gitlab

def get_first_project_name(gitserver, keyword):
    projects = gitserver.projects.list(owned=True,search=keyword)
    if len(projects) > 0:
        return projects[0].name
    return None

# example code showing how this all works.
if __name__=="__main__":

# NB: the second field is an API access token.  It has now expired.
# Sorry, aspiring hackers.
```

[6] https://docs.python.org/dev/library/unittest.mock-examples.html

```
    gl = gitlab.Gitlab('http://gitlab.com','Q6ypEE8QbqdeUaf7TAq5' )

    gl.auth()

    print(get_first_project_name(gl, "Team"))
```

So, let's write a unit test for this function, without using any mocking:

```
import unittest
import gitlab
from gitlabutilities import get_first_project_name

class UtilitiesTest(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls._gitlab =
gitlab.Gitlab('http://gitlab.com','Q6ypEE8QbqdeUaf7TAq5')
        cls._gitlab.auth()

    def test_basic(self):
        name = get_first_project_name(self.__class__._gitlab, "Team")
        self.assertEqual(name, "Team-TheGitLabMafia")
```

One thing about this code that you might not have seen before is the use of `setUpClass` rather than `SetUp`. The difference between the two is that `setUpClass` only runs *once* for each test class. In this case it doesn't make a whole lot of difference, as there's only one test method in the class, but if we had more test methods, this could potentially save some time.

Anyway, does our test work? Yes...but look at the execution time:

```
xubuntu@xubuntu-VirtualBox:~/gdrive/teaching/teaching resources/FIT2107
Lecture Notes/mocking_code$ python3 gitlab-utilities-test.py
.
----------------------------------------------------------------------
Ran 1 test in 4.169s
```

```
OK
xubuntu@xubuntu-VirtualBox:~/gdrive/teaching/teaching resources/FIT2107
Lecture Notes/mocking_code$
```

Furthermore, this test works when you use an access token that grants access to my GitLab account. On a different account - or even the same account some time in the future where the repositories in it have changed - the results will be different.

Therefore, the calls to the `python-gitlab` library are prime candidates for some mocking.

```python
import unittest
from unittest.mock import MagicMock
import gitlab
from gitlabutilities import get_first_project_name

class UtilitiesTest(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls._gitlab = MagicMock()
        gitlabmafia = MagicMock()
        gitlabmafia.name ="Team-TheGitLabMafia"
        cls._gitlab.projects.list.return_value=[gitlabmafia]

    def test_basic(self):
        name = get_first_project_name(self.__class__._gitlab, "Team")
        self.assertEqual(name, "Team-TheGitLabMafia")

if __name__ == "__main__":
    unittest.main()
```

The first thing to note is that the `gitlabutilities` code itself has not changed. That's the *point* of mocking. If you find yourself having two versions of the code you're trying to test - one for testing, and one for production use, ***YOU ARE DOING IT WRONG.***

Our `setUpClass` creates *three* `MagicMocks`. Yes, three. Two of them are created explicitly, and one is created automatically in the last line, where we access

`cls._gitlab.projects.list`. In `get_first_project_name`, we call `gitlab.projects.list` to retrieve the list of projects, so we need to set a return value - and the return value we set is a list which contains a single `Mock` object, upon which we have set a `name` field. If we didn't set a `name` field, when it's accessed in the test, it would have created yet another `mock` object, which is NOT what we want!

The test itself is identical - at this point. We're not attempting to check whether the API we're mocking is called correctly.

And the results? It's… a tad quicker :)

```
xubuntu@xubuntu-VirtualBox:~/gdrive/teaching/teaching resources/FIT2107
Lecture Notes/mocking_code$ python3 gitlab-utilities-test-mocked.py
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

Our very simple mocking example has solved several problems with unit testing this method:

- The test now has *predictable* results
- The test doesn't require secret information (access to my GitLab account) to run.
- The test runs orders of magnitude faster than before.

What it does not, however, do is check that the mocked method has been called correctly.

Checking how mocks are called

Mock objects keep track of the arguments with which they have been called. You can check these later using your choice of a set of assertion methods. You can choose to simply check whether the mock has been called, or to check precisely how many times, and with what arguments, it has been called. In this case, we know that we should query the list object precisely once, and we know exactly what arguments it should be called with. We can then specify that with the assertion split over the last two lines:

```
def test_basic(self):
        name = get_first_project_name(self.__class__._gitlab, "Team")
```

```
        self.assertEqual(name, "Team-TheGitLabMafia")
        self.__class__._gitlab.projects.list.\
             assert_called_once_with(owned=True, search='Team')
```

It's worth noting here that just because you *can* check precisely how a mock has been called, it doesn't mean that you always *should*.

So why wouldn't you precisely check how a mock has been called?

It's often a lot of work to write comprehensive assertions to check the mock calls, and that work may be made redundant if you refactor the code under test to use the thing you've mocked in a different way to get the same result. This is not only a waste of effort, it means that your unit test is *less useful* than it should have been, because the unit test can't be used to check that the *externally-visible behavior*, which is the thing you care about, is maintained through the refactor!

So when should you check the mock calls? Unfortunately, guidelines for this are kinda thing on the ground, but from what I've found:

- You should check anything that is part of the externally-specified behaviour of the module. For example, if the requirements say that your should open a file and write certain things to it, and you're mocking the file access, you need to check that the calls to the file-access methods are made and are correct.
- You should check things if it's the only convenient way to verify the correctness of the code you're testing. In the case of `test-basic()`, there is no other way to check that the right information is being fished out of the gitlab API other than to check the contents of the calls.
- You don't have to check every call to a mock in every single unit test. For instance, the single test I've done is probably sufficient to check that the mocked method is getting the right arguments; I wouldn't bother checking it again in other tests of `get_first_project_name`.
- Otherwise, think very carefully whether you need to check the correctness of a particular call to a mock. Generally, unit tests should have one or two assertions, not twenty, and checking every call to every mock object is going to violate this rule left right and center.

Speccing your mocks

No, not that kind of speccie (Photo: Erin Phillips takes a spectacular high mark (a "speccie") in the inaugural AFL Women's Grand Final. Source: The Age[7])

One of the properties of the `Mock` family is that if an attribute doesn't exist, it will be created for you unless you explicitly delete it (with the `del` operator). If you use a `Mock` or `MagicMock`, those attributes will even be callable:

```
>>> from unittest.mock import Mock
>>> x=Mock()
>>> print(x.y())
<Mock name='mock.y()' id='139997368724336'>
>>> if x.z():
...     print("hello")
...
hello
>>> del x.a
>>> x.a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.5/unittest/mock.py", line 587, in __getattr__
    raise AttributeError(name)
AttributeError: a
```

[7]http://www.theage.com.au/afl/afl-match-report/aflw-grand-final-glory-to-the-adelaide-crows-thanks-to-a-perfect-10-from-erin-phillips-20170325-gv6ezl.html

```
>>> x.a()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.5/unittest/mock.py", line 587, in __getattr__
    raise AttributeError(name)
AttributeError: a
>>>
```

This feature is extremely useful, but can also mean a lot of bugs that should get caught by testing are not.

For instance, consider our assertion to check that the mock is called correctly. What happens if we misspell the assertion, so our test looks like this?

```
   def test_basic(self):
        name = get_first_project_name(self.__class__._gitlab, "Team")
        self.assertEqual(name, "Team-TheGitLabMafia")
        self.__class__._gitlab.projects.list.\
        asert_called_once_with(owned=True, search='Team')
```

Unfortunately, the test succeeds despite the typo:

```
xubuntu@xubuntu-VirtualBox:~/gdrive/teaching/teaching resources/FIT2107
Lecture Notes/mocking_code$ python3
gitlab-utilities-test-checkmocked-namebug.py
.
----------------------------------------------------------------------
Ran 1 test in 0.004s

OK
```

OK, let's insert a bug in the method call that the assertion was supposed to check.

```
def get_first_project_name(gitserver, keyword):
    projects = gitserver.projects.list(pwned=True, search=keyword)
    if len(projects) > 0:
        return projects[0].name
```

```
    return None
```

Now, let's run our buggy test…

```
xubuntu@xubuntu-VirtualBox:~/gdrive/teaching/teaching resources/FIT2107
Lecture Notes/mocking_code$ python3
gitlab-utilities-test-checkmocked-namebug.py
.
----------------------------------------------------------------------
Ran 1 test in 0.002s

OK
```

No valid assertion...bug goes undetected!

Generally, you want your mock objects to only support the same attributes and methods that the real object that it's substituting for, and raise exceptions otherwise. *Speccing* a mock allows you to impose that restriction.

There are a number of ways to spec a mock, as described in the official Mock documentation, but one way to do so is to create a real object of the type you want, and spec the mock based on the real object using `create_autospec()`:

```
@classmethod
    def setUpClass(cls):
        realgitlab = gitlab.Gitlab('fake@fake.com', "test")
        cls._gitlab = create_autospec(realgitlab)
        gitlabmafia = MagicMock()
        gitlabmafia.name = "Team-TheGitLabMafia"
        cls._gitlab.projects.list.return_value = [gitlabmafia]
```

THIS IS NOT THE ONLY WAY TO SPEC A MOCK, AND IT MAY NOT BE THE BEST WAY TO DO SO FOR YOUR OWN CODE. READ THE OFFICIAL DOCUMENTATION ON AUTOSPEC!

In short, what this does is spec the mock based on the real attributes of the object, so that if you try to reference an attribute of `cls._gitlab` that didn't exist in `realgitlab`, you'll get an AttributeError. Furthermore, the mocks created for the same attributes *work the same way* - that is, if you try an access a non-existent attribute for `cls._gitlab.projects`, you'll also get

an error - and this applies to the mocks that are the attributes of the attributes of `cls._gitlab`, and so on.

The downside of this is that we've had to create a real `gitlab.Gitlab` object (which might be slow, expensive, and require the very network/disk/whatever access we're trying to avoid) to copy from. Why couldn't `create_autospec` work off the class definition - why did it need an actual object?

The problem is that it's common practice to create new attributes in the `__init__` method of a Python class, so they do not exist in the class definition (boo dynamic typing…). So, if we wish to avoid creating real objects to copy the mock from, either we have to manually add additional attributes to the `Mock` object we create manually, or do some other clever thing. If the code you are planning to mock is your own, appropriate design can make autospeccing easier - for instance, by setting default values attributes in the class definition where autospec can find them. This aspect of `mock` is less clean than I - and the `mock` library authors - would like, and it's possible that some aspects of it may change in the future.

However, once you've done the work of speccing your mocks, it can catch all manner of errors, both in your code *and* your tests. For instance, our misspelled assertion now gets caught by the Python runtime!

```
xubuntu@xubuntu-VirtualBox:~/gdrive/teaching/teaching resources/FIT2107
Lecture Notes/mocking_code$ python3 gitlab-utilities-test-bugautospec.py
E
======================================================================
ERROR: test_basic (__main__.UtilitiesTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "gitlab-utilities-test-bugautospec.py", line 18, in test_basic
    self.__class__._gitlab.projects.list.\
  File "/usr/lib/python3.5/unittest/mock.py", line 578, in __getattr__
    raise AttributeError("Mock object has no attribute %r" % name)
AttributeError: Mock object has no attribute 'asert_called_once_with'

----------------------------------------------------------------------
Ran 1 test in 0.021s

FAILED (errors=1)
```

Win!

Patching

We now return to the situation where we need to mock an object or method that we don't have access to directly.

Here's a modified version of our function:

```
import gitlab

def get_first_project_name(serverURL, token, keyword):
    '''Note: this is not  good design - this function is
    even less cohesive than it was previously.
    It's a way to demonstrate patching - no more, no less!
    '''
    gitserver = gitlab.Gitlab(serverURL, token)
    gitserver.auth()
    projects = gitserver.projects.list(owned=True,search=keyword)
    if len(projects) > 0:
        return projects[0].name
    return None


if __name__ == "__main__":

print(get_first_project_name('http://gitlab.com','Q6ypEE8QbqdeUaf7TAq5',"Te
am"))
```

Now, we have a problem. We still want to mock the `gitlab.Gitlab` object, but our test has no access to it as it's created within `get_first_project_name()`. So, rather than creating a `mock` object and passing it it in, we need to patch!

```
class UtilitiesTest(unittest.TestCase):

    @patch('gitlab.Gitlab')
    def test_basic(self, classmock):
        gitlabmafia = MagicMock()
        gitlabmafia.name = "Team-TheGitLabMafia"
        gitlabmock = MagicMock()
        gitlabmock.projects.list.return_value = [gitlabmafia]
        classmock.return_value = gitlabmock
```

```
        name = get_first_project_name("https://gitlab.com", "test-token",
"Team")
        self.assertEqual(name, "Team-TheGitLabMafia")
        gitlabmock.projects.list.assert_called_once_with(owned=True,
search="Team")
```

This case is an example of patching a *class*. You can patch a single method or an object using `patch`, but in this case we need to patch every access to `gitlab.Gitlab` objects and the simplest way to do so is by patching the class.

The patch replaces the *class* with a `MagicMock` object, which gets passed into `test_basic()` as an additional argument. If you apply multiple patches, they will be added as further arguments in reverse order (so the last patch is the first additional argument, the second-last patch is the second additional argument, and so on).

If we want to control what an *instance* of the class is like, we need to set the return value for the class mock. In this case, we'll simply make the class mock return a `MagicMock`, as before. We then set up a return value for the relevant instance method on the attribute of the attribute of the mock that actually gets called

Note: the details of this stuff can get mind-numbingly tedious, but it's important to understand what's going on here. Work through the code and make sure you understand what every line is doing!

Finally, we invoke our function and see whether the results are as expected, as we've done in other versions of our unit test.

Does it work? Yes!

```
xubuntu@xubuntu-VirtualBox:~/gdrive/teaching/teaching resources/FIT2107 Lecture
Notes/mocking_code$ python3 gitlab-utilities-test-mockedv2.py
.
----------------------------------------------------------------------
Ran 1 test in 0.002s

OK
```

Some of you may have noticed that we haven't used `autospec()` to create our mocks. There's a reason for that. If we patch `gitlab.Gitlab` for the entire test function, we don't have any access to it to create a real `gitlab.Gitlab` object that the autospecced mock can copy.

What we can do, however, is use `patch` and the `with` statement[8] to apply the patch to a part of the test method:

```
    def test_basic(self):
        realgitlab = gitlab.Gitlab("https://gitlab.com", "faketoken")
        with patch('gitlab.Gitlab') as classmock:
            gitlabmafia = MagicMock()
            gitlabmafia.name = "Team-TheGitLabMafia"
            gitlabmock = create_autospec(realgitlab)
            gitlabmock.projects.list.return_value = [gitlabmafia]
            classmock.return_value = gitlabmock

            name = get_first_project_name("https://gitlab.com",
"test-token", "Team")
            self.assertEqual(name, "Team-TheGitLabMafia")
            gitlabmock.projects.list.assert_called_once_with(owned=True,
search="Team")
```

The code hasn't changed much, except that we create a real `gitlab.Gitlab` object, *then* patch gitlab.Gitlab, then use the real object to create an appropriately autospecced mock `gitlab.Gitlab` instance.

One final wrinkle - targeting your patches

Most of the time, you don't really have to think too hard about the effect of `import` statements on Python name resolution. Unfortunately, when you're trying to mess with the natural order of things with patching, this stuff can matter.

Let's change the way we import `gitlab.Gitlab` into the code we're trying to test to use `from...import` rather than a straight `import`:

```
from gitlab import Gitlab

def get_first_project_name(serverURL, token, keyword):
    gitserver = Gitlab(serverURL, token)
    gitserver.auth()
```

[8] "Context managers" are a very cool feature of Python that handle cleaning up things after you've finished with them for you: https://jeffknupp.com/blog/2016/03/07/python-with-context-managers/

```
    projects = gitserver.projects.list(owned=True,search=keyword)
    if len(projects) > 0:
        return projects[0].name
    return None
```

if we make this simple change, patching `gitlab.Gitlab` in our test code no longer works:

```
xubuntu@xubuntu-VirtualBox:~/gdrive/teaching/teaching resources/FIT2107
Lecture Notes/mocking_code$ python3 gitlab-utilities-test-mockedv4.py
E
======================================================================
ERROR: test_basic (__main__.UtilitiesTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "gitlab-utilities-test-mockedv4.py", line 19, in test_basic
    name=get_first_project_name("https://gitlab.com", "test-token", "Team")
  File "/media/sf_Google_Drive/teaching/teaching resources/FIT2107 Lecture
Notes/mocking_code/gitlabutilities3.py", line 5, in get_first_project_name
    gitserver.auth()
  File "/usr/local/lib/python3.5/dist-packages/gitlab/__init__.py", line
195, in auth
    self._token_auth()
  File "/usr/local/lib/python3.5/dist-packages/gitlab/__init__.py", line
230, in _token_auth
    self.user = self._objects.CurrentUser(self)
  File "/usr/local/lib/python3.5/dist-packages/gitlab/base.py", line 387,
in __init__
    data = self.gitlab.get(self.__class__, data, **kwargs)
  File "/usr/local/lib/python3.5/dist-packages/gitlab/__init__.py", line
497, in get
    raise_error_from_response(r, GitlabGetError)
  File "/usr/local/lib/python3.5/dist-packages/gitlab/exceptions.py", line
222, in raise_error_from_response
    response_body=response.content)
gitlab.exceptions.GitlabAuthenticationError: 401: 401 Unauthorized

----------------------------------------------------------------------
Ran 1 test in 1.054s

FAILED (errors=1)
```

It's trying to use the real Gitlab library code, not the mock! What should we do?

As the mock official documentation explains, *"The key is to patch out `SomeClass` (ed: the thing you're trying to patch) where it is used (or where it is looked up)."*

When you use from..import, the thing we've imported is imported into the namespace of the module we're running. So, in this case, rather than patching `gitlab.Gitlab`, we need to patch <our module name>`.Gitlab`:

```
   def test_basic(self):
        realgitlab = gitlab.Gitlab("https://gitlab.com", "faketoken")
        # gitlabutilities3.py is the name of the module we've written
        with patch('gitlabutilities3.Gitlab') as classmock:
            gitlabmafia = MagicMock()
            gitlabmafia.name = "Team-TheGitLabMafia"
            gitlabmock = create_autospec(realgitlab)
            gitlabmock.projects.list.return_value = [gitlabmafia]
            classmock.return_value = gitlabmock

            name = get_first_project_name("https://gitlab.com",
"test-token", "Team")
            self.assertEqual(name, "Team-TheGitLabMafia")
            gitlabmock.projects.list.assert_called_once_with(owned=True,
search="Team")
```

And, voila, everything works again:

```
Notes/mocking_code$ python3 gitlab-utilities-test-mockedv4.py
.
----------------------------------------------------------------------
Ran 1 test in 0.019s

OK
```

Digression: This is hard!

The mock library does some rather complex things, and consequently requires a fairly deep delve into the internals of Python to understand and use.

Some of you might be tempted to think "what's the point - this mocking stuff is more trouble than it's worth". And, frankly, sometimes it is.

However, it's worth persisting and trying to get your head around the complexities for several reasons:

- Mocking is a genuinely useful tool - it allows you to test things you can't efficiently test otherwise.
- Understanding the library properly will actually allow you to use it correctly - so that you can trust the results of your testing - and efficiently, so that your unit tests can run quickly enough to use repeatedly when you're in the middle of development or bugfixing.
- In the process of understanding this, you'll learn more about Python's class and module system, which will be useful elsewhere. I know I did.
- Finally, when you're out in the workforce, you're going to have to learn about other complex tools and methodologies. One of the things we're trying to teach you at university is how to learn new things without having them spoonfed to you - but, if you get stuck, you have a lecturer, tutors, *and* fellow students who can help. Learn to learn now, and it will be easier next time!

Mocking best practices

In this chapter, we've concentrated mainly on the *how* of mocking, when you've made the decision to mock a particular class or function. What we haven't spent as much time talking about is when you should you mock, and what should you mock, and when you should seek alternatives.

Useful recommendations are, unfortunately, a bit thin on the ground, but these guidelines by Gil Zilberfeld and Dror Helper are a good place to start[9]. Translated into Python-English, they are:

1. **Know what to isolate**: look at the guidelines from the start of these notes about when mocking is useful. If they don't apply, don't bother. Even then consider whether an alternative approach might be better.
2. **Fake as little as needed**: The more you have to fake, the more chances of having a hard-to-find bug in your faking, and the more "brittle" the tests become.

[9] http://www.methodsandtools.com/archive/archive.php?id=122

3. **Fake the immediate neighbors:** Patching a method call several steps down the call tree is technically possible, but very bug-prone and, again, "brittle", because any change in any one of the intermediate classes might cause the mocking to not work right.
4. **Don't misuse assertions on the arguments to mocked methods:** In the original version this was "don't misuse verify", which serves a similar purpose in the Mockito mocking framework for Java. As noted earlier in the notes, you should generally check the arguments for mock calls only to a) verify correctness when there's no other way to do so, or b) it forms part of the external definition of the module's correct behavior.
5. **One or two assertions per test:** To reduce the possibilities of your tests having errors, and to make it easier to understand what's going on within a test to use the results, they should be as simple as possible. Generally, one test should be only checking one or two things rather than a laundry list. This ties neatly back into point 4.

An additional couple of useful suggestions that I've found:

- **Try to avoid mocking other people's code where possible**. This sounds a bit rich, given that the major example in this chapter does precisely that. However, other people's interfaces are subject to change beyond your control, and are often quite difficult to mock well (as, indeed, we saw in the examples, which was great for demonstrating the full capabilities of `mock` but not so easy to understand). Rather than mocking other people's (or, more to the point, other organizations') code, at least consider whether you should have an abstraction layer which you can easily mock.
- **If your code is hard to mock, that's a code smell.** Generally, if your interfaces are simple, share minimal information, and aren't "chatty" (that is, there are complex sequences of calls to do anything) mocking will be relatively easy. Guess what - those same things make using those interfaces easy as well. So if mocking an interface is awkward, it's highly likely that the code that uses it will be more awkward than it needs to be, so think about redesigning the interface. This is an extension of the general principle that testable code is likely to be high-quality code more generally.

Alternatives to using `mock`

While `mock`, and mocking libraries more generally, can make testing easier, they are not always the best way to test a particular piece of code. Sometimes, it may make more sense to use an alternative method, including:

- hand-roll your own test doubles rather than using the `mock` library.
- If you're mocking something that calls a network service, set up your own version of the service for testing. For instance, rather than mocking gitlab, you set up your own gitlab server, with known test data, on the same local network as the machines that run the tests.

- If you're mocking to avoid user interactions, consider using a tool like Selenium to automate the user interactions, rather than using mocks in the tests themselves.

Some of these solutions may be more appropriate for integration testing or system testing than unit testing, but sometimes they might be right for your problem.

For what it's worth, for the web application I developed, MADAM, I chose *not* to mock the database interactions in our unit tests. This means the tests run a lot slower than they would if I mocked them, but it avoids writing a lot of complex code to check that the database API calls are correct. We can actually see, directly, whether the desired changes in database state are being made, and we can also inspect the results of the database changes back through the API to see if they're working right. For us, this was a better option, given just how much of MADAM is simply about putting things into and getting things out of a database.

Remember, the goal of functional correctness testing is to *reveal deviations from the specified function of the system* as effectively and efficiently as possible. It's your job as a developer to use the right testing toolchain to do that. Mocking libraries are a powerful tool that you can use , but they are not the only one.

Chapter summary

- Sometimes, when testing an individual class or module within a system, we want to substitute code in other parts of the system with "test doubles" that we can more easily control and monitor.
- While we can often "hand-roll" test doubles in Python, it can involve a fairly deep dive into the language internals, and can get tedious and bug-prone.
- The `mock` library simplifies the process of creating test doubles, members of the Mock class hierarchy.
- `Mock` objects can be easily configured to return appropriate data from attributes and methods.
- `Mock` objects can also report how they are used by the code you're trying to test.
- By default, any reference to an attribute or method on a `Mock` object that doesn't yet exist, results in another `Mock` being created. This can lead to hard-to-detect problems in your test. Speccing your `Mock` objects reduces the risk of these problems.
- You can use patching to mock objects and methods that are created locally within the code under test.
- Mocking is a powerful tool, but it is not a cure-all. Use it wisely, and sometimes alternative approaches may be more appropriate.

Recommended reading

- General tutorials
    - https://www.toptal.com/python/an-introduction-to-mocking-in-python
    - https://semaphoreci.com/community/tutorials/getting-started-with-mocking-in-python
    - https://blog.fugue.co/2016-02-11-python-mocking-101.html
- The following two tutorials are interesting in that they demonstrate some alternative ways to use the mock library
    - http://www.drdobbs.com/testing/using-mocks-in-python/240168251
    - http://www.drdobbs.com/architecture-and-design/patching-mocks-in-python/240168924
- https://medium.com/python-pandemonium/python-mocking-you-are-a-tricksy-beast-6c4a1f8d19b2 - good discussion about mocking class vs instance methods.
- http://www.methodsandtools.com/archive/archive.php?id=122 - introduction to mocking in Java. Source of our discussion of best practices.