Why your mock doesn't work

Friday 2 August 2019

Mocking is a powerful technique for isolating tests from undesired interactions among components. But often people find their mock isn't taking effect, and it's not clear why. Hopefully this explanation will clear things up.

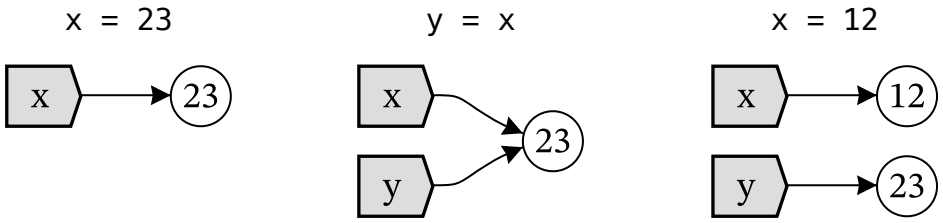
BTW: it's really easy to over-use mocking. These are good explanations of alternative approaches:

- Augie Fackler and Nathaniel Manista's PyCon talk Stop Mocking and Start Testing.
- Itamar Turner-Trauring's article Fast tests for slow services: why you should use verified fakes.

A quick aside about assignment

Before we get to fancy stuff like mocks, I want to review a little bit about Python assignment. You may already know this, but bear with me. Everything that follows is going to be directly related to this simple example.

Variables in Python are names that refer to values. If we assign a second name, the names don't refer to each other, they both refer to the same value. If one of the names is then assigned again, the other name isn't affected:



If this is unfamiliar to you, or you just want to look at more pictures like this, Python Names and Values goes into much more depth about the semantics of Python assignment.

Importing

Let's say we have a simple module like this:

```
# mod.py

val = "original"

def update_val():
```

```
    global val
    val = "updated"
```

We want to use val from this module, and also call update_val to change val. There are two ways we could try to do it. At first glance, it seems like they would do the same thing.

The first version imports the names we want, and uses them:

```
# code1.py

from mod import val, update_val

print(val)
update_val()
print(val)
```

The second version imports the module, and uses the names as attributes on the module object:

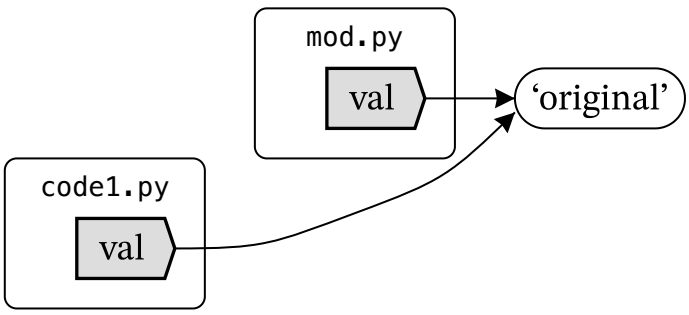
```
# code2.py

import mod

print(mod.val)
mod.update_val()
print(mod.val)
```

This seems like a subtle distinction, almost a stylistic choice. But code1.py prints “original original”: the value hasn’t changed! Code2.py does what we expected: it prints “original updated.” Why the difference?

Let’s look at code1.py more closely:

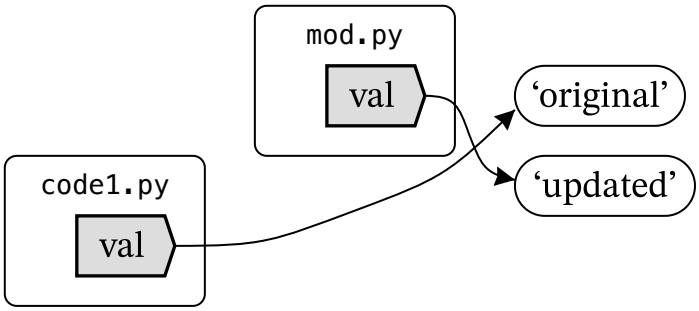
```
# code1.py

from mod import val, update_val

print(val)
update_val()
print(val)
```

After “from mod import val”, when we first print val, we have this:



“from mod import val” means, import mod, and then do the assignment “val = mod.val”. This makes our name val refer to the same object as mod’s name val.

After “update_val()”, when we print val again, our world looks like this:

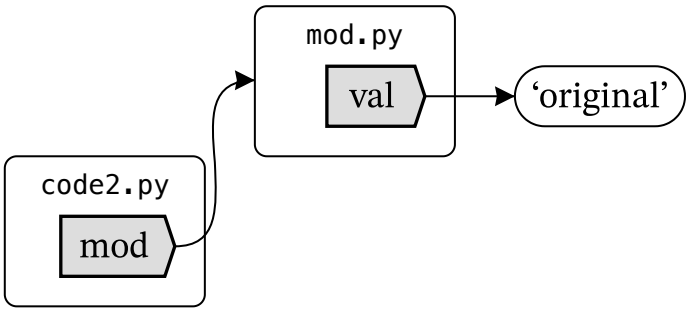


update_val has reassigned mod’s val, but that has no effect on our val. This is the same behavior as our x and y example, but with imports instead of more obvious assignments. In code1.py, “from mod import val” is an assignment from mod.val to val, and works exactly like “y = x” does. Later assignments to mod.val don’t affect our val, just as later assignments to x don’t affect y.

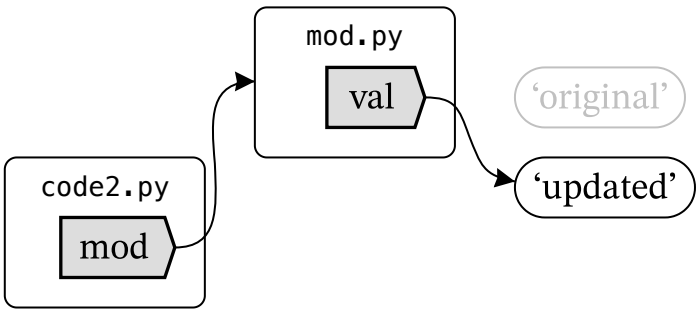
Now let’s look at code2.py again:

```
# code2.py

import mod

print(mod.val)
mod.update_val()
print(mod.val)
```

The “import mod” statement means, make my name mod refer to the entire mod module. Accessing mod.val will reach into the mod module, find its val name, and use its value.



Then after “update_val()”, mod’s name val has been changed:



Now we print mod.val again, and see its updated value, just as we expected.

OK, but what about mocks?

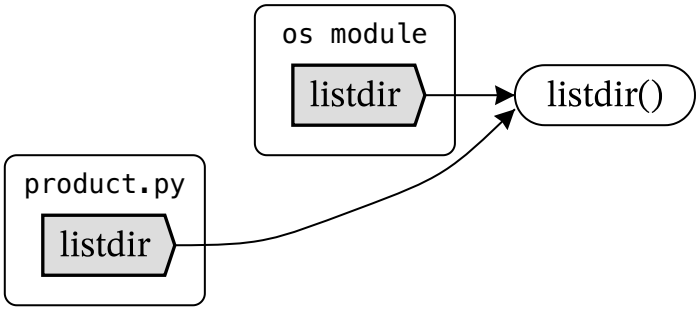
Mocking is a fancy kind of assignment: replace an object (or function) with a different one. We'll use the mock.patch function in a with statement. It makes a mock object, assigns it to the name given, and then restores the original value at the end of the with statement.

Let's consider this (very roughly sketched) product code and test:

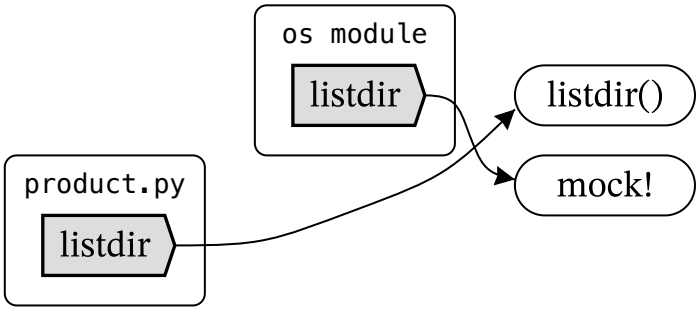
```
# product.py

from os import listdir

def my_function():
    files = listdir(some_directory)
    # ... use the file names ...
```

```
# test.py

def test_it():
    with mock.patch("os.listdir") as listdir:
        listdir.return_value = ['a.txt', 'b.txt', 'c.txt']
        my_function()
```

After we've imported product.py, both the os module and product.py have a name "listdir" which refers to the built-in listdir() function. The references look like this:



The mock.patch in our test is really just a fancy assignment to the name "os.listdir". During the test, the references look like this:



You can see why the mock doesn't work: we're mocking something, but it's not the thing our product code is going to call. This situation is exactly analogous to our code1.py

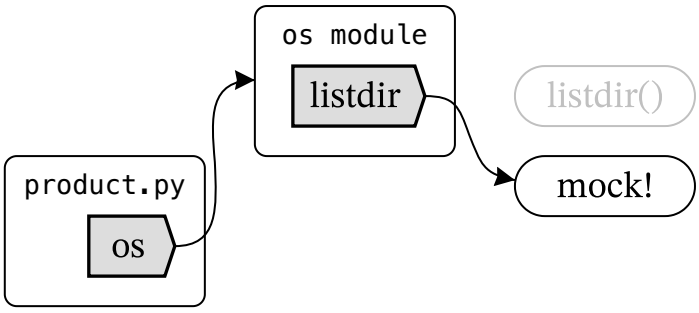
example from earlier.

You might be thinking, “ok, so let’s do that code2.py thing to make it work!” If we do, it will work. Your product code and test will now look like this (the test code is unchanged):

```
# product.py

import os

def my_function():
    files = os.listdir(some_directory)
    # ... use the file names ...
```

```
# test.py

def test_it():
    with mock.patch("os.listdir") as listdir:
        listdir.return_value = ['a.txt', 'b.txt', 'c.txt']
        my_function()
```

When the test is run, the references look like this:



Because the product code refers to the os module, changing the name in the module is enough to affect the product code.

But there’s still a problem: this will mock that function for any module using it. This might be a more widespread effect than you intended. Perhaps your product code also calls some helpers, which also need to list files. The helpers might end up using your mock (depending how they imported os.listdir!), which isn’t what you wanted.

Mock it where it’s used

The best approach to mocking is to mock the object where it is used, not where it is defined. Your product and test code will look like this:

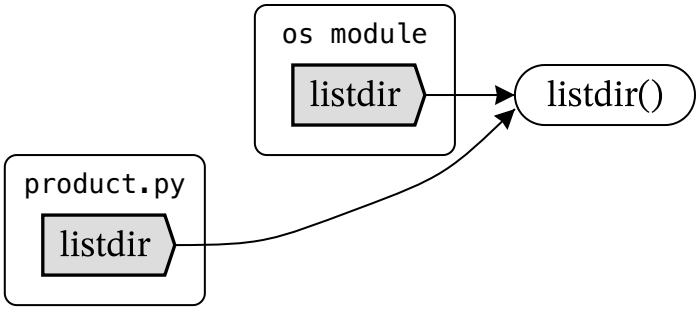
```
# product.py

from os import listdir

def my_function():
    files = listdir(some_directory)
    # ... use the file names ...
```

```python
# test.py

def test_it():
    with mock.patch("product.listdir") as listdir:
        listdir.return_value = False
        my_function()
```

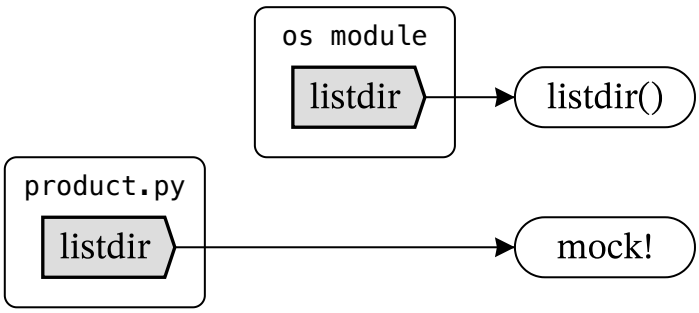
The only difference here from our first try is that we mock “product.listdir”, not “os.listdir”. That seems odd, because listdir isn’t defined in product.py. That’s fine, the name “listdir” is in both the os module and in product.py, and they are both references to the thing you want to mock. Neither is a more real name than the other.

By mocking where the object is used, we have tighter control over what callers are affected. Since we only want product.py’s behavior to change, we mock the name in product.py. This also makes the test more clearly tied to product.py.

As before, our references look like this once product.py has been fully imported:



The difference now is how the mock changes things. During the test, our references look like this:



The code in product.py will use the mock, and no other code will. Just what we wanted!

Is this OK?

At this point, you might be concerned: it seems like mocking is kind of delicate. Notice that even with our last example, how we create the mock depends on something as arbitrary as how we imported the function. If our code had “import os” at the top, we wouldn’t have been able to create our mock properly. This is something that could be changed in a refactoring, but at least mock.patch will fail in that case.

You are right to be concerned: mocking is delicate. It depends on implementation details of the product code to construct the test. There are many reasons to be wary of mocks, and there are other approaches to solving the problems of isolating your product code from problematic dependencies.

If you do use mocks, at least now you know how to make them work, but again, there are other approaches. See the links at the top of this page.

» 16 reactions *#python #testing*
