UNIVERSITY *of* WASHINGTON

Software Design for Data Science

Advanced Testing

Melissa Winstanley
University of Washington
February 22, 2024

Types of testing

- Unit testing
- ……

Types of testing

- Unit testing
- Integration testing
- Validation testing
- System/E2E testing
- UI testing
- Regression testing
- Black box/white box testing
- Fuzzy testing
- Performance testing
- Usability testing
- Security testing
- Accessibility testing
- Load testing
- Continuous testing

Today

Mocking

UI testing in Streamlit

Mocking

Mocking: motivation

- Your code makes an HTTP request to a website, which may return data that varies over time.
- How to write a test that validates…
  - …that your code works if there are HTTP errors?
  - …that your code works if the data changes?
  - …that your code works in normal cases?
- …given that you don’t control the network or the other website?

Mocking: motivation

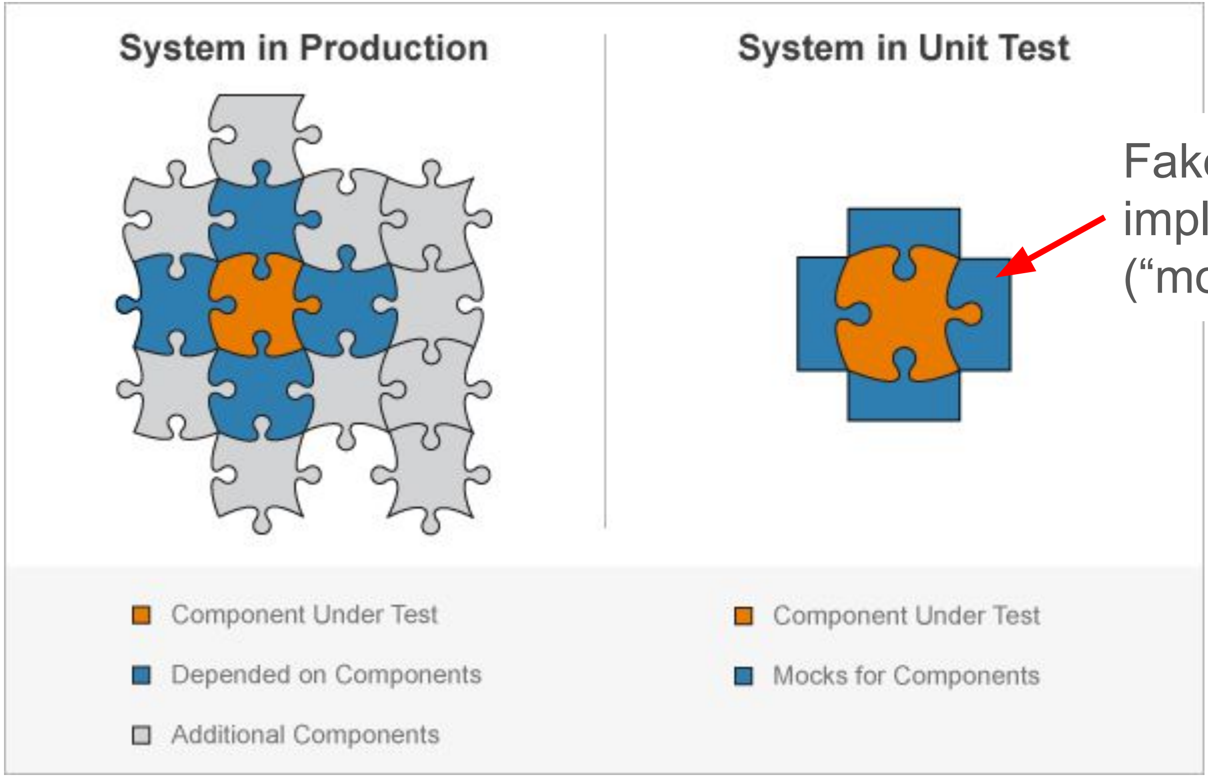
Another example: Homework 3

- In HW3, you wrote `check_email_validity` and `process_newsgroup_file`.
- The instructors want to validate your code - but we want to grade each function separately!
    - If your `check_email_validity` is wrong, you should not ALSO lose points for `process_newsgroup_file`, assuming nothing else is wrong.
- One solution: run OUR implementation of `check_email_validity` with YOUR implementation of `process_newsgroup_file`
- How can we do that?

Mocking: the problem

How do we unit test our code when it depends on other functions, modules, or input that we may not have control over?

Mocking



Fake implementations (“mocks”)

Image source: https://www.mathworks.com/help/matlab/mocking-framework.html

Mocking in Python

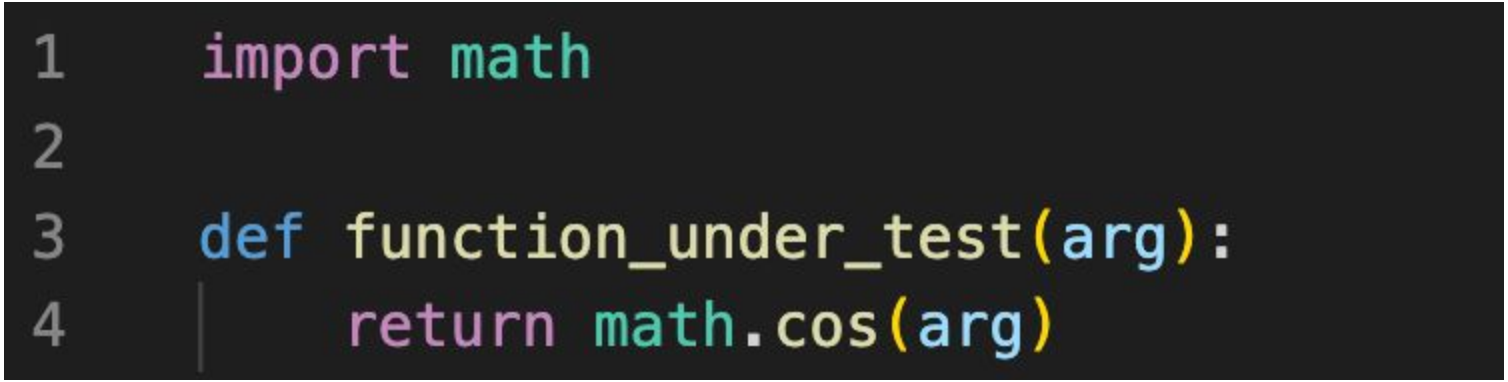
The unittest library has a module for mocking

```
from unittest import mock
```

https://docs.python.org/3/library/unittest.mock.html

Allows you to provide false *implementations*, *return values*, *exceptions*, and more for a particular function, module, or class.

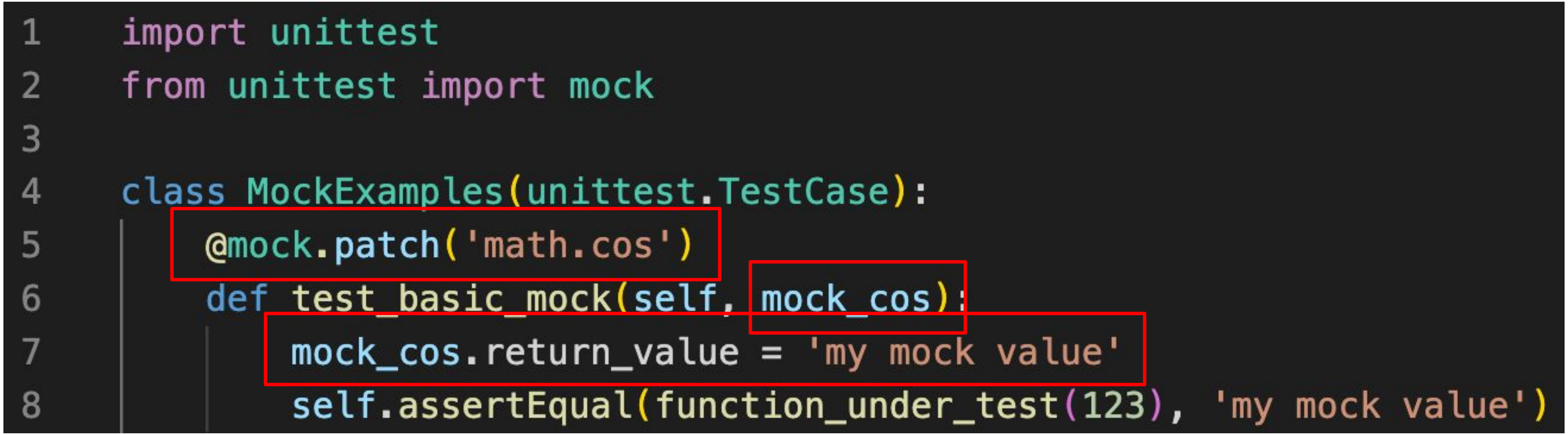
Example: the code under test



```python
import math

def function_under_test(arg):
    return math.cos(arg)
```

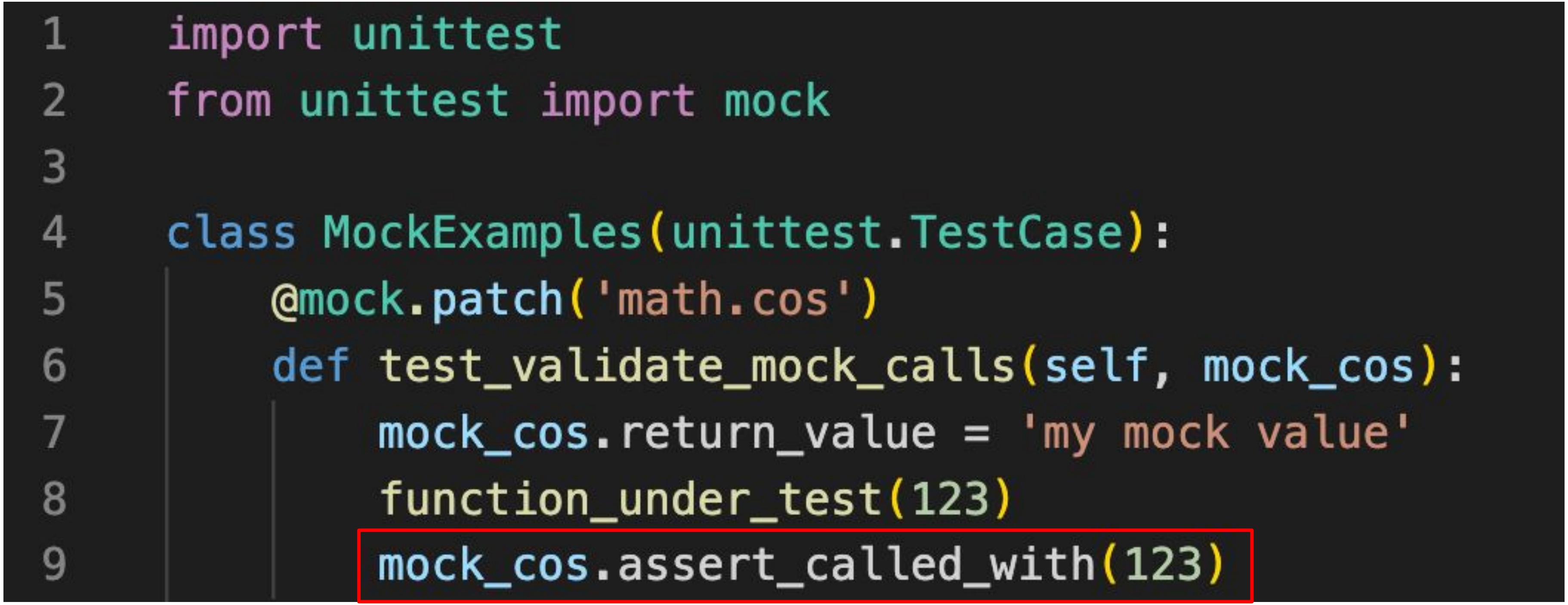
Using the `@mock.patch` decorator



```python
import unittest
from unittest import mock

class MockExamples(unittest.TestCase):
    @mock.patch('math.cos')
    def test_basic_mock(self, mock_cos):
        mock_cos.return_value = 'my mock value'
        self.assertEqual(function_under_test(123), 'my mock value')
```

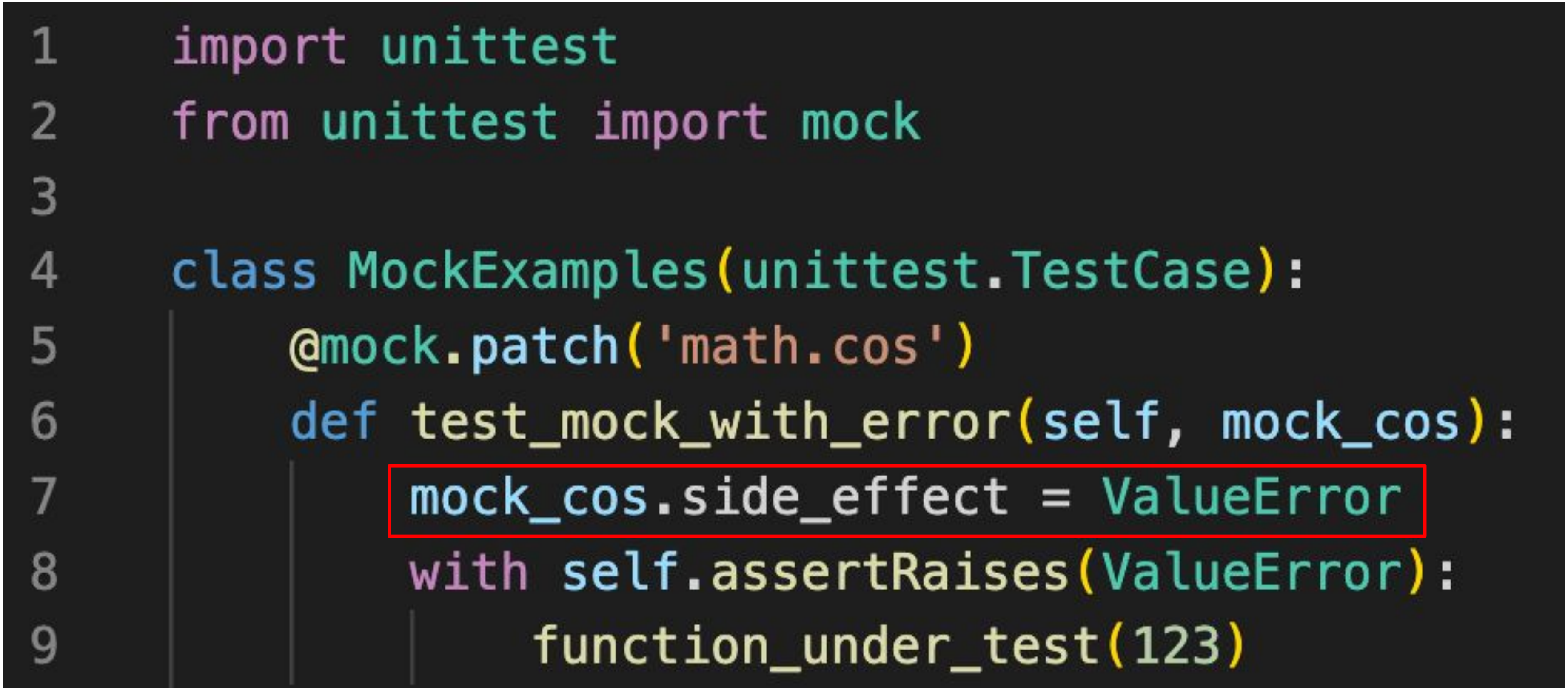
Using `assert_called_with`



```python
import unittest
from unittest import mock

class MockExamples(unittest.TestCase):
    @mock.patch('math.cos')
    def test_validate_mock_calls(self, mock_cos):
        mock_cos.return_value = 'my mock value'
        function_under_test(123)
        mock_cos.assert_called_with(123)
```

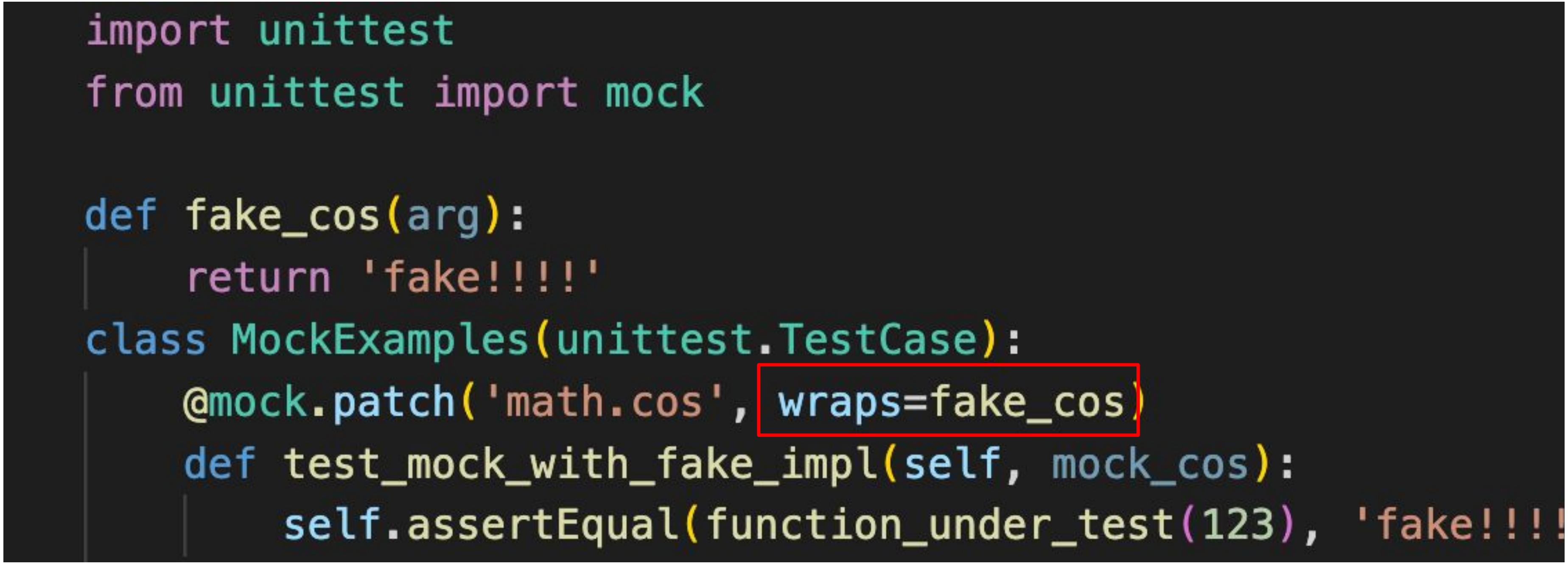
Using `side_effect` to trigger an exception



```python
import unittest
from unittest import mock

class MockExamples(unittest.TestCase):
    @mock.patch('math.cos')
    def test_mock_with_error(self, mock_cos):
        mock_cos.side_effect = ValueError
        with self.assertRaises(ValueError):
            function_under_test(123)
```

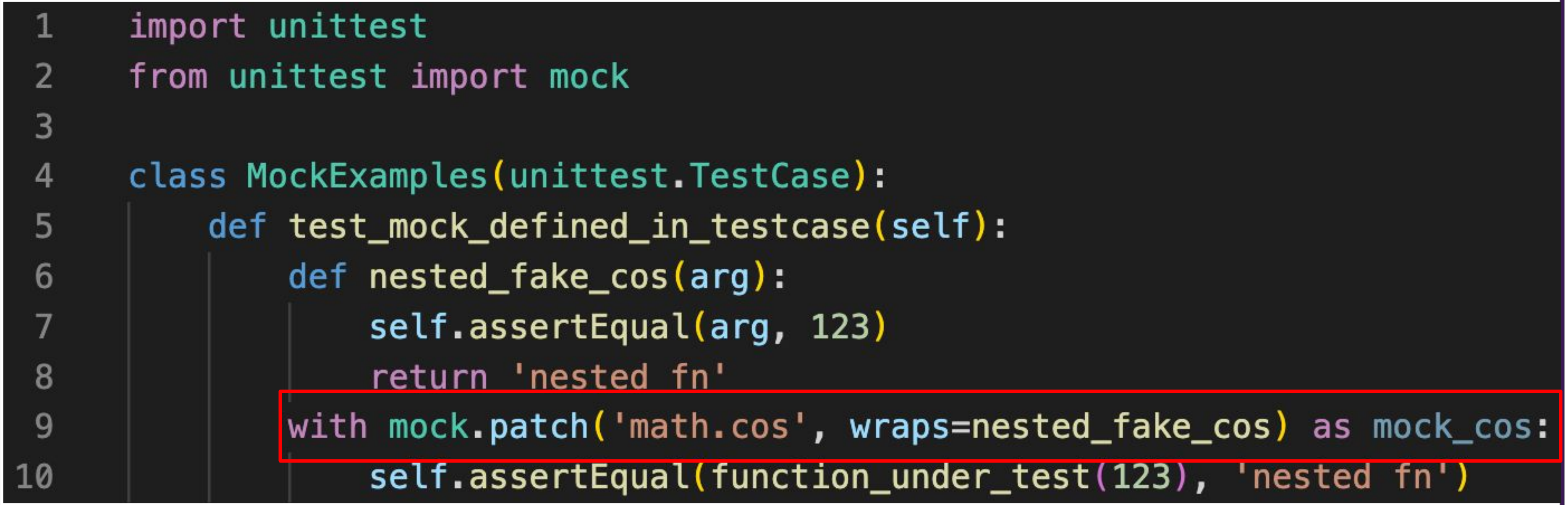
Wrapping a fake implementation



```
import unittest
from unittest import mock

def fake_cos(arg):
    return 'fake!!!!'
class MockExamples(unittest.TestCase):
    @mock.patch('math.cos', wraps=fake_cos)
    def test_mock_with_fake_impl(self, mock_cos):
        self.assertEqual(function_under_test(123), 'fake!!!!
```

Nesting a mock to use `self` or other variables



```python
import unittest
from unittest import mock

class MockExamples(unittest.TestCase):
    def test_mock_defined_in_testcase(self):
        def nested_fake_cos(arg):
            self.assertEqual(arg, 123)
            return 'nested fn'
        with mock.patch('math.cos', wraps=nested_fake_cos) as mock_cos:
            self.assertEqual(function_under_test(123), 'nested fn')
```

Exercise: Mocking

- From the Syllabus, clone the linked repository

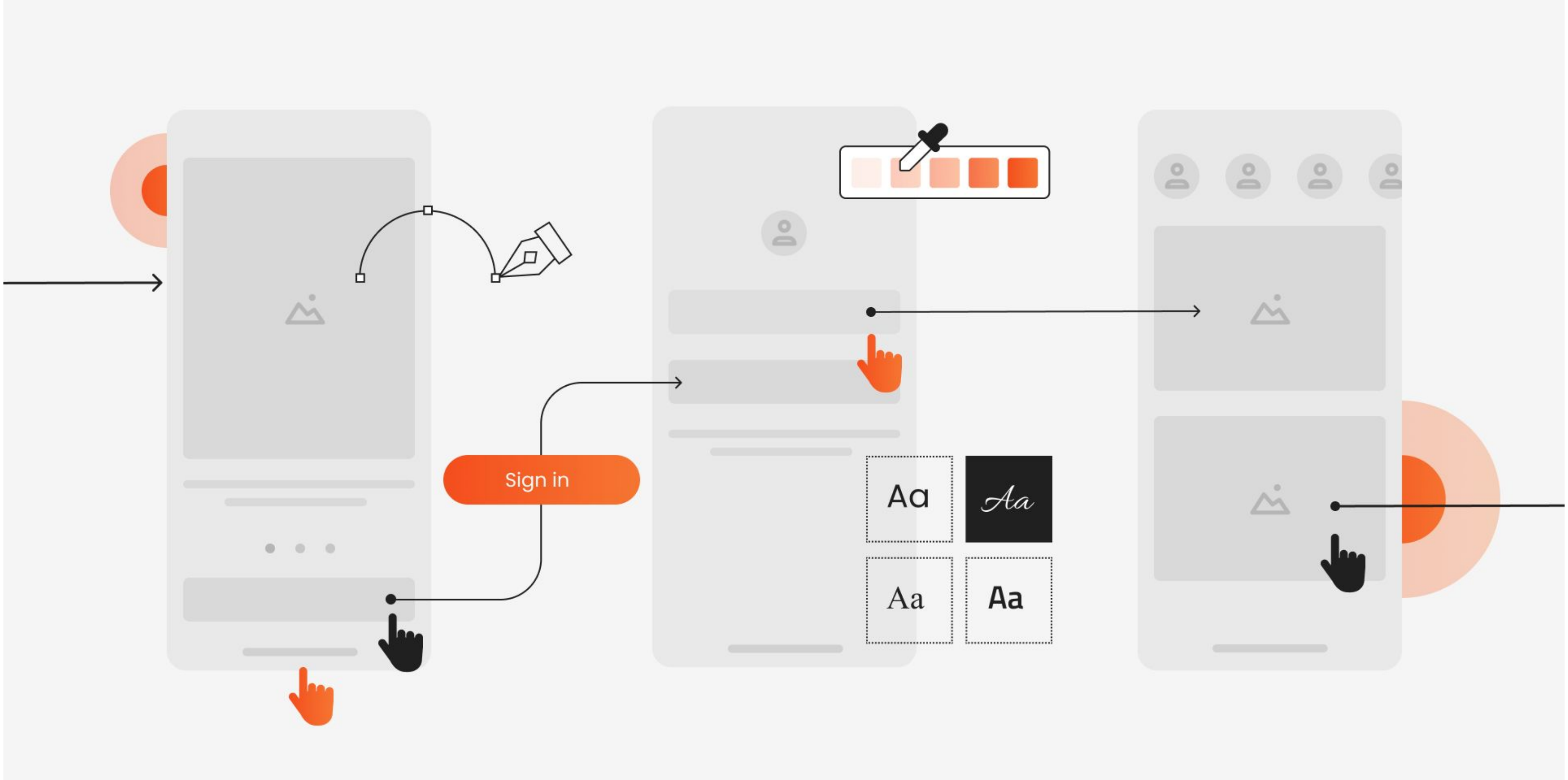
```
git clone https://github.com/UWDATA515/testing_example.git
```

- Cd into the mocking_example directory
- Look at the README and follow the directions to try mocking

UI Testing

What is UI testing?

- Test the user interface
  - GUI
  - CLI
- Specifically, the visual elements, layout, and interactions
- Lots of tools, but they get pretty complicated
  - Selenium
  - Cypress


Sign in
Aa
Aa
Aa
Aa

UI testing

How to use Streamlit’s AppTest

In your test function, create an “AppTest”:

```
at = AppTest.from_file("app.py").run()
```

Instead of importing the file normally, use `from_file`

Assert that different content is what you expect it to be:

```
self.assertEqual(at.text[0].value,"Whatever the content should be")
```

Access UI elements by their type through the app test object. Here, [0] signifies that there might be many text elements, and we want the first one

Interact with components:

```
at.button[0].click().run()
```

Different interactive UI elements have different behaviors that you can trigger

Exercise: Streamlit testing

- In the same testing repository as before
- Cd into the ui_testing_example directory
- Look at the README and follow the directions to try adding tests

If you aren’t using Streamlit, feel free to use this opportunity to research what testing options exist for your user interface framework.