In this lecture I’m going to go over how you can make sure you’re running your unit test in isolation using the concepts of Dummies, Fakes, Stubs, Spies, and Mocks.

* So what are test doubles?
* Almost all code that gets implemented will depend on another piece of code in the system.
* Those other pieces of code are often times trying to do things or communicate with things that are not available in the unit testing environment, or are so slow that they would make our unit tests extremely slow. For example, if you’re code queries a 3rd party REST API on the internet and that server is down for any reason you can’t run your tests.
* Test doubles are the answer to that problem. They are objects created in the test to replace the real production system collaborators.
* There are many types of test doubles.
* Dummy objects are the simplest. They are simply placeholders that are intended to be passed around but not actually called or used in any real way. They will often generate exceptions if they are called.
* Fake objects have a diﬀerent (and usually simplified) implementation from the production collaborator that make them useable in the test code but not suitable for production.
* Stubs provide implementations that do expect to be called but respond with basic canned responses.
* Spies provide implementations that record the values that are passed in to them. The tests can then use those recorded values for validating the code under test.
* Mock objects are the most sophisticated of all the test doubles. They have pre-programmed expectations about the ordering of calls, the number of times functions will be called, and the values that will be passed in. Mock objects will generate their own exceptions when these pre-programmed expectations are not met.
* Mock frameworks are libraries that provide easy to use API’s for automatically creating any of these types of test doubles AT RUNTIME.
* They provide easy API’s for specifying the mocking expectations in your unit tests.
* They can be much more eﬃcient than implementing your own custom mock objects.
* As creating your own custom mock objects can be time consuming, tedious, and error prone.
* unittest.mock is a mocking framework for Python.
* It’s built-in to the standard unittest library for Python version 3.3 and newer.
* For older versions of Python a backported version of the library is available on PyPi called mock and can be installed with the command “pip install mock”.

unittest.mock - Mock Class

•

# Example

def test\_Foo(): bar = Mock()

functionThatUsesBar( bar )

bar.assert\_called\_once()

unittest.mock provides the Mock

class which can be used as a fake, stub, spy, or true mock for all your tests.

•

The Mock class has many

initialization parameters for controlling its behavior.

* Once it has been called a Mock

object has many built-in functions for verifying how it was used.

* Unittest.mock provides the Mock class which is an extremely power class that be used to create test objects that can be used as fakes, stubs, spies, or true mocks for other classes or functions.
* The Mock class has many initialization parameters for specifying how the object should behave such as what interface it should mock, if it should call another function

when it is called, or what value it should return.

* Once a Mock object has been used it has many built-in functions for verifying how it was used such as how many times it was called and with what parameters.

Mock - Initialization

•

# Example

def test\_Foo():

bar = Mock(spec=SpecClass) bar2= Mock(side\_effect=

barFunc)

bar3 = Mock(return\_value=1)

•

•

•

Mock provides many initialization

parameters which can be used to control the mock objects behavior.

The “spec” parameter specifies the interface that Mock object is implementing.

The “side\_effect” parameters specifies a function that should be called when the mock is called.

The “return\_value” parameter specifies the return value when the Mock is called.

* Mock provides many initialization parameters which can be used to control the mock object’s behavior.
* The spec parameter specifies the interface that the Mock object is implementing. If any attributes of the mock object are called which are not in that interface then the Mock will automatically generate an AttributeError exception.
* The side\_eﬀect parameter specifies a function that should be called when the mock is called. This can be useful for more complicated test logic that returns diﬀerent values depending on input parameters or generates exceptions.
* The return\_value parameter specifies the value that should be returned when the mock object is called. If the side\_eﬀect parameter is set it’s return value is used instead.
* Mock provides many built-in functions for verifying how the mock was called including the following assert functions.
* The assert\_called function will pass if the mock was ever called with any parameters.
* The assert\_called\_once function will pass if the mock was called exactly once.
* The assert\_called\_with function will pass if the mock was last called with the specified parameters.
* The assert\_called\_once\_with function will pass if the mock was called exactly once with the specified parameters.
* The assert\_any\_call function will pass if the mock was ever called with the specified parameters
* And the assert\_not\_called function will pass if the mock was never called.
* Mock provides these additional built-in attributes for verifying how it was called.
* The assert\_has\_calls function passes if the mock was called with the parameters specified in each of the passed in list of mock call objects and optionally in the order that those calls are put in the array.
* The called attribute is a boolean which is true if the mock was ever called.
* The call\_count attribute is an integer value specifying the number of times the mock object was called.
* The call\_args attribute contains the parameters that the mock was last called with.
* The call\_args\_list attribute is a list with each entry containing the parameters that were used in a call to the mock object.
* Unittest.mock also provides the MagicMock class.
* MagicMock is derived from Mock and provides a default implementation of most of the Python magic methods. These are the methods with double undressores at the beginning and end of the name like str and int .
* The following magic names are not supported by MagicMock due to being used by Mock for other things or because mocking them could cause other issues: getattribute, setattribute, init, new, prepare, instancecheck, subclass check, and delete.
* I will use MagicMock by default in all of the examples in this course. I also use it by default in practice as it can simplify test setup. When using MagicMock you just need to keep in mind the fact that the magic methods are already created and take note of the default values that are returned from those functions to ensure they match the needs of the test that’s being implemented.

PyTest Monkeypatch Test Fixture

•

def callIt()

print(“Hello World”)

PyTest provides the monkeypatch

test fixture to allow a test to dynamically replace:

def test\_patch(monkeypatch)

monkeypatch(callIt, Mock()) callIt()

callIt.assert\_called\_once()

* module and class attributes

•

Dictionary entries

•

Environment Variables

* PyTest provides the monkeypatch test fixture to allow a test to dynamically change:
* Module and class attributes
* Dictionary entries
* And Environment Variables
* Unittest provides a patch decorator which performs similar operations but this can sometimes conflict with the PyTest TestFixture decorators so I’ll focus on using monkeypatch for this functionality.
* In the next lecture I’ll go over several examples of using Mock and Monkeypatch in diﬀerent test scenarios.