Test Driven Development(TDD) Using Python

INSTRUCTOR: Richard Wells
https://gale.udemy.com/course/unit-testing-and-tdd-in-python/learn/lecture/8419010#overview

Section 1: Introduction

Trying to keep your code base clean from bugs is a never ending task. Having a buggy code base can cause a lot of problems. It can affect your schedule and some portion of your development team has to stop new feature development to go back and fix the bugs. It can slow down development in general as developers will be more wary to make changes when they are working in a buggy and brittle code base. And it can erode your customers' confidence both because they are experiencing the bugs and because of the terrible code schedule.

What can we as a software developer do to mitigate these problems? We need to have a multilayer safety net of tests in place that will catch any bugs that might get introduced. And the first layer of this safety net should be a suite of automated unit tests. Unit tests verify your code at the level of functions and classes. They perform positive and negative tests at the lowest level of your code. Every line of your production code should have an associated unit test that verifies that it is working as expected.

Test Driven Development or TDD is a practice of writing Unit Tests before writing your code. This seems backwards but it helps in many ways. You will know every line of production code is working as soon as it is written because you are testing it immediately. If there is a problem it is easy to track down, and TDD gives you and your team the confidence to change the code as you will know immediately if anything breaks due to the particular code change.

We will see now what TDD is, where TDD came from and how TDD is dome with python.

Section 2: Overview of Unit Testing and Test Driven Development

What is Unit Testing

Why do we Unit Test?

- Software bugs hurt the business
- Software testing catches the bugs before they get to the field
- Done systematically with a multilayered approach where each layer of testing provides several levels of safety nets to catch the bugs

Levels of testing

- Unit tests - Testing at a functional level
- Component Testing - Tests the external interface for individual components (Components are essentially a collection of the functions)
- System Testing - TEsts the external interfaces of a system which is a collection of subsystems
- Performance testing - And lastly comes performance testing, which tests systems and subsystems that expected production loads to verify that response times and resource utilization, i.e. memory, CPU and disk usage are acceptable.

Unit Testing Specifics

- Unit testing tests individual functions in the code
- Each function should have a corresponding Unit Test
- Groups of Unit Tests can be combined into Test Suites which can help with organizations
- Unit tests should be run in a Development environment rather than a Production environment. This enables us to run them easily and often
- Unit Tests should be implemented in an Automated fashion

We can see an example where the test_string_length() call is a Unit Test for the str_len(thestr) Production Code

The Unit Test performs 3 steps

- SetUp where it creates the Test string
- Action where it calls the Production function to test the string
- Assertion weather the test validates the results of the Action

What is Test Driven Development

What is Test Driven Development?

- TDD is a process where you take personal responsibility for the quality of your code
- We do this by writing the Unit Tests before the production code
- Even though the tests are written before the production code, we do not write all the test or all the production code at once
- The testing and the production are written together in small bits of functionality

What are some of the benefits of TDD

- TDD gives you the confidence to make changes in the code because you have to test before you begin and verify that the code is working and will tell you of any of your changes have broken anything
- This confidence comes from the immediate feedback the tests provide
- The tests document what the production code is supposed to do and a new developer can use the Unit Tests as a sort of documentation
- Writing a good test can drive good Object Oriented Design as making individual classes and functions testable in isolation drives the developer to break the dependencies and add interface rather than linking concrete implementations

TDD Workflow: Red, Green, Refactor

TDD has the following phases in a workflow

- Red Phase - The red phase is to write a failing unit test for the next bit of functionality you want to implement in the in the production code
- Green Phase - Where you write just enough Production Code for which the Unit Test passes
- Refactor Phase - Where you clean up the Unit Test and the Production Code to remove any duplication to make sure the code follows your teams coding standards and best practices
- Repeat the process for all the functionality you need to implement and all the positive and negative test cases

Uncle Bobs 3 Laws of TDD

Robert Martin created the following laws of TDD in his book 'Clean Code - A Handbook of Agile Software Development'. These laws help you follow the TDD process

1. You may not write any production code until you have written a failing Unit Test
2. You may not write more of a Unit Test than is sufficient to fail, and not compiling is failing (this makes our unit tests very simple)
3. You may not write more Production Code than is sufficient to pass the currently failing unit test case.

Example of the TDD Session - The FizzBuzz Kata

Green Phase - Write Failing Unit Test

```
def test_FizzBuzz():
   assert FizzBuzz(1) == 1
```

Red Phase - Write enough production code to pass the test

```
def FizzBuzz(value):
   return value
```

Refactor Phase - None for now

Next Phase - We need a string of 2 when 2 in passed in

Green Phase -

```
def test_FizzBuzzTwo():
   retVal = FizzBuzz(2)
   assert retVal == "2"
```

Red Phase -

```
def FizzBuzz(value):
   return str(value)
```

Refactor Phase -

We have

```
def test_FizzBuzz():
   assert FizzBuzz(1) == "1"


def test_FizzBuzzTwo():
   retVal = FizzBuzz(2)
   assert retVal == "2"
```

We can rewrite as:

```
def test_FizzBuzz(value, expected):
   retval = FizzBuzz(value)
   assert retval == expected

def test_FizzBuzzOne():
   test_FizzBuzz(1, "1")

def test_FizzBuzzTwo():
   test_FizzBuzz(2, "2")
```

Next Phase 3

Green Phase

```
def test_FizzBuzzThree():
   test_FizzBuzz(3,"Fizz")
```

Red Phase

```
def FizzBuzz(value):
   if value == 3:
       value = "Fizz"
   return str(value)
```

Next Phase 5

Green Phase

```
def test_FizzBuzzFive():
   test_FizzBuzz(5, "Buzz")
```

Red Phase

```
def FizzBuzz(value):
   if value == 3:
       value = "Fizz"
   if value == 5:
       value = "Buzz"
   return str(value)
```

And so on

Section 4: PyTest Overview

PyTest Overview

What is PyTest?

- PyTest is a unit testing framework
- It provides the ability to create Tests, Test Modules, and Test Fixtures
- Uses the built-in Python assert statement
- Has Command Line parameters to help filter which tests are executed and in what order

Creating a Test

- Test functions need to have the word “test” at the beginning
- Test do the verification using standard=d python assert statement.
- Tests can be grouped together in the same module or class

```
(venv) C:\Users\aksha\PycharmProjects\TDD>pytest -v

============================= test session starts ==============================
platform win32 -- Python 3.11.4, pytest-7.4.0, pluggy-1.3.0 --
C:\Users\aksha\PycharmProjects\TDD\venv\Scripts\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\aksha\PycharmProjects\TDD
collected 7 items

test_test.py::test_FizzBuzzOne PASSED                                    [ 14%]
test_test.py::test_FizzBuzzTwo PASSED                                    [ 28%]
test_test.py::test_FizzBuzzThree PASSED                                  [ 42%]
test_test.py::test_FizzBuzzFive PASSED                                   [ 57%]
test_test.py::test_FizzBuzzSix PASSED                                    [ 71%]
test_test.py::test_FizzBuzzTen PASSED                                    [ 85%]
test_test.py::test_FizzBuzzFift PASSED                                   [100%]
============================== 7 passed in 0.02s ==============================
```

Test Discovery

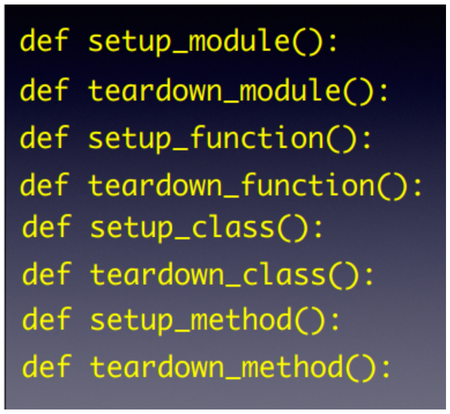
We will now discuss PyTests ability to automatically discover Unit Tests

Naming Convention

- Pytest will automatically discover tests when you execute based on standard naming convention.
- Test functions should start with “test at the beginning.
- Classes with tests in them should have “Test at the beginning of the class name and not have and “__init__” method
- The file names of test modules should start or end with “test”. (ex. Test_example.py or example_test.py)

XUnit Style Setup and Teardown

One key feature of all unit test frameworks is providing the ability to execute setup code before and after the test (test modules, test functions, test Classes, test methods in test classes). PyTest provides this capability with both XUnit style setup and tear down functions and with Pytest fixtures.



```
def setup_module():
def teardown_module():
def setup_function():
def teardown_function():
def setup_class():
def teardown_class():
def setup_method():
def teardown_method():
```

Using the setup and teardown functions can help reduce code duplication by letting you specify the setup and teardown code once at each of the different levels as necessary rather than repeating the code in each individual unit test. This can help keep the code clean and manageable.

In the Examples we notice that PyTest will call on each SetUp function before each unit test in that module that is not in the class. It will then call the teardown function after each unit test has completed executing. SetUp function and teardown function are passed in the unit test that is being executed so the setup and teardown code can be customized per unit test if necessary.

```python
def setup_function(function):
    if function == test1:
        print("\nSetting up test 1")
    elif function == test2:
        print("\nSetting up test 2")
    else:
        print("\nSetting up unknown test")

def teardown_function(function):
    if function == test1:
        print("\nTearing up test 1")
    elif function == test2:
        print("\nTearing up test 2")
    else:
        print("\nTearing up unknown test")

def test1():
    print("Executing test 1")
    assert True

def test2():
    print("Executing test 2")
    assert True
```

In terminal:

```
pytest -v -s
```

-v for verbose output

-s for no console output and only print statements

Output:

```
================================= test session starts =================================
platform win32 -- Python 3.11.4, pytest-7.4.0, pluggy-1.3.0 --
C:\Users\aksha\PycharmProjects\TDD\venv\Scripts\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\aksha\PycharmProjects\TDD
collected 2 items

test_test.py::test1
Setting up test 1
Executing test 1
PASSED
Tearing up test 1

test_test.py::test2
Setting up test 2
Executing test 2
PASSED
Tearing up test 2

=================================2 passed in 0.01s =================================
```

Now let's extend this example to include setup and teardown functions for the module itself. These functions will be called once before any of the unit tests in the module are executed and again after all the unit tests in the module are executed. Now we update the code

```
def setup_module(module):
   print("\nSetup module")

def teardown_module(module):
   print("\nTeardown module")
```

Output:

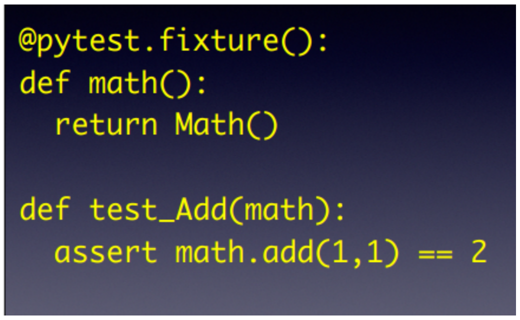
```
=================================== test session starts ===================================
platform win32 -- Python 3.11.4, pytest-7.4.0, pluggy-1.3.0 --
C:\Users\aksha\PycharmProjects\TDD\venv\Scripts\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\aksha\PycharmProjects\TDD
collected 2 items

test_test.py::test1
Setup module

Setting up test 1
Executing test 1
PASSED
Tearing up test 1

test_test.py::test2
Setting up test 2
Executing test 2
PASSED
Tearing up test 2

Teardown module

=================================== 2 passed in 0.02s ===================================
```

In another example we have moved test 1 and test 2 unit test functions into a unit class called Test class. The setup_method and teardown_method has the @classmethod Decorator applied as they are passed in the uninstantiated class object rather than a unique instance of the class.

```
class TestClass:

   @classmethod
   def setup_class(cls):
       print("\nSetup class")

   @classmethod
   def teardown_class(cls):
       print("\nTeardown class")

   def setup_method(self, method):
       if method == self.test1:
           print("\nSetting up test 1")
       elif method == self.test2:
           print("\nSetting up test 2")
       else:
           print("\nSetting up unknown test")

   def teardown_method(self, method):
       if method == self.test1:
           print("\nTearing up test 1")
       elif method == self.test2:
           print("\nTearing up test 2")
       else:
           print("\nTearing up unknown test")

   def test1(self):
       print("Executing test 1")
       assert True

   def test2(self):
       print("Executing test 2")
       assert True
```

Test Fixtures

In this lecture we're going to give you an overview of PyTest Fixtures.We'll explain how Test Fixtures differ from the classic XUnit style setup and teardown functions and we'll go over some examples.



```
@pytest.fixture():
def math():
  return Math()

def test_Add(math):
  assert math.add(1,1) == 2
```

Like the XUnit style of setup and teardown functions, Test fixtures allow for re-use of code across tests by specifying functions that should be executed before the unit test runs.

We can specify that a function is a test fixture by applying the pytest.fixture decorator to the function.

Individual unit tests can specify they want to use that function by specifying it in their parameter list or by using the pytest.mark.usefixture decorator.

The fixture can also set its autouse parameter to true which will cause all tests in the fixture's scope to automatically execute the fixture before the test executes.

```
import pytest

@pytest.fixture()
def setup():
   print("\nSetup")

def test1():
   print("Executing test 1")
   assert True

def test2():
   print("Executing test 2")
   assert True
```

Output:

collected 2 items

```
test_test.py::test1 Executing test 1
PASSED
test_test.py::test2 Executing test 2
PASSED
```

We see that the setup does not work like this

```python
@pytest.fixture()
def setup():
   print("\nSetup")

def test1(setup):
   print("Executing test 1")
   assert True

@pytest.mark.usefixtures("setup")
def test2():
   print("Executing test 2")
   assert True
```

Output:

```
collected 2 items

test_test.py::test1
Setup
Executing test 1
PASSED
test_test.py::test2
Setup
Executing test 2
PASSED
```

Often there is some type of teardown or cleanup that a test, class, or module needs to perform after testing has been completed. Each test fixture can specify their own teardown code that should be executed. There are two methods of specifying a teardown code for a test fixture: The yield keyword and the request-context object's addfinalizer method.

Yield

- The yield keyword is the simpler of the two options for teardown code. The code after the yield is executed after the fixture goes out of scope.
- The yield keyword is a replacement for return and any return values should be passed to it.

```
@pytest.fixture():
def setup():
        print("Setup!")
        yield
        print("Teardown!")
```

Addfinalizer

- The addfinalizer method of adding teardown code is a little more complicated but also a little more capable than the yield statement.
- With the addfinalizer method one or more finalizer functions are added via the request-context's addfinalizer method.
- One of the big differences between this method and the yield keyword method is that this method allows for multiple finalization functions to be specified.

```
@pytest.fixture():
def setup(request):
        print("Setup!")
def teardown:
        print("Teardown!")
request.addfinalizer(teardown)
```

```
@pytest.fixture()
def setup1():
   print("\nSetup1")
   yield
   print("\nTeardown")

@pytest.fixture()
def setup2(request):
   print("\nSetup2")
   def teardown_a():
       print("\nTeardown A")
   def teardown_b():
       print("\nTeardown B")
   request.addfinalizer(teardown_a)
   request.addfinalizer(teardown_b)

def test1(setup1):
   print("Executing test 1")
   assert True

def test2(setup2):
   print("Executing test 2")
   assert True
```

Scope

Which tests a test fixture applies to and how often it is run depends on the fixture's scope. Test fixtures have four different scopes: By default the scope is set to function and this specifies that the fixture should be called for all tests in the module. Class scope specifies the test fixture should be executed once per test class. Module scope specifies that the fixture should be executed once per module. Session scope specifies that the fixture should be executed once when PyTest starts.

- Function - Run the fixture once for each test
- Class - Run the fixture once for each class of tests
- Module - Run once when the module goes in scope
- Session - The fixture is run when pytest starts.

```
import pytest

@pytest.fixture(scope="module", autouse=True)
def setupModule2():
   print("\nSetup Module")

@pytest.fixture(scope="class", autouse=True)
def setupClass2():
   print("\nSetup Class")

@pytest.fixture(scope="function", autouse=True)
def setupFunction2():
   print("\nSetup Function")

class TestClass:
   def test1(self):
       print("Executing test 1")
       assert True
   def test2(self):
       print("Executing test 2")
       assert True
```

PyTest Test Fixtures allow you to optionally return data from the fixture that can be used in the test. The optional params array argument in the fixture decorator can be used to specify one or more values that should be passed to the test. When a params argument has multiple values then the test will be called once with each value. Let's look at a working example

```
@pytest.fixture(params=[1, 2, 3])
def setup(request):
   retVal = request.param
   print("\nSetup, retVal = {}".format(retVal))
   return retVal

def test(setup):
   print("\nsetup = {}".format(setup))
   assert True
```

Assert Statements and Exceptions

Pytest allows the use of the built in python assert statement for performing verifications in a unit test. The normal comparison operators can be used on all python data types: less than, greater than, less than or equal, greater than or equal, equal, or not equal. Pytest expands on the messages that are reported for assert failures to provide more context in the test results.

```
def test_IntAssert():
    assert 1 == 1
def test_StrAssert():
    assert “str” == “str”
def test_floatAssert():
    assert 1.0 == 1.0
def test_arrayAssert():
    assert [1,2,3] == [1,2,3]
def test_dictAssert():
    assert {“1”:1} == {“1”:1}
```

Validating floating point values can sometimes be difficult as internally the value is stored as a series of binary fractions. Because of this some comparisons that we’d expect to pass will fail. Pytest provides the approx function which will validate that two floating point values are “approximately” the same value as each other to within a default tolerance of 1 time E to the -6 value.

```
# Failing Test!!!
def test_BadFloatCompare():
    assert (0.1 + 0.2) == 0.3

# Passing Test!!!
def test_GoodFloatCompare():
    val = 0.1 + 0.2
    assert val == approx(0.3)
```

In some test cases we need to verify that a function raises an exception under certain conditions. - Pytest provides the raises helper to perform this verification using the “with” keyword. - When the “raises” helper is used the unit test will fail if the specified exception is not thrown in the code block after the “raises line

```
def test_Exception():
        with raises(ValueError)
                raise ValueError
```

PyTest Command Line Arguments

By default Pytest runs all tests that it finds in the current working directory and sub-directory using the naming conventions for automatic test discovery.

There are several pytest command line arguments that can be specified to try and be more selective about which tests will be executed.

- You can simply pass in the module name to execute only the unit tests in one particular module.
- You can also simply pass in a directory path to have pytest run only the tests in that directory.
- You can use the “-k” option to specify an evaluation string based on keywords such as: module name, class name, and function name.
- You can use the “-m” option to specify that any tests that have a “pytest.mark” decorator that matches the specified expression string will be executed.

Here are some additional command line arguments that can be very useful.

- The -v option specifies that verbose output from pytest should be enabled.
- The -q option specifies the opposite. It specifies that the tests should be run quietly (or minimal output). This can be helpful from a performance perspective when you’re running 100’s or 1000’s of tests.
- The -s option specifies that PyTest should NOT capture the console output.
- The —ignore option allows you to specify a path that should be ignore during test discovery.
- The —maxfail option specifies that PyTest should stop after N number of test failures

Section 5: The Supermarket Checkout Kata

Overview

Checkout Class that maintains a list of items that are being checked out.

Checkout Class provides interfaces for:

- Setting the price of individual items
- Adding individual items to a checkout
- The current total cost for all the items added
- Add and apply an optional discount on select items when N number are purchased

Test Cases

- Can create an instance of the Checkout class
- Can add an item price
- Can add an item
- Can calculate the current total
- Can add multiple items and get correct total
- Can add discount rules
- Can apply discount ruled to the total
- Exception is thrown when item is added which does not have a defined price

SetUp and First Test Case

Let's do the (1st) Green Phase

Checkout.py

```
class Checkout:
   pass
```

TestCheckout.py

```
from Checkout import *
def test_instantiateCheckout():
   co = Checkout()
```

Add Items, Add Item prices, and Calculate Current Total

Red Phase

Let's make sure the test exists and fails

```
from Checkout import *

def test_instantiateCheckout():
   co = Checkout()

def test_AddItemPrice():
   co = Checkout()
   co.addItemPrice('a', 1)
```

We now have a failing test (red phase)

Green Phase

(make sure all tests pass for the most minimal amount of code):

Checkout.py

```
class Checkout:

   def addItemPrice(self, item, price):
       pass
```

Refactoring Phase

What is Important? We notice checkout instantiation has been done twice. Let’s change that.

Red Phase

Back to Red Phase for the next point. We need to update the checkout class so that we can have a public interface to add checkout items.

```
def test_canAddITem():
    co = Checkout()
    co.addItem("a")
```

Green Phase

Let's make it work:

In Checkout.py

```
class Checkout:
    def addItemPrice(self, item, price):
        pass
    def addItem(self, item):
        pass
```

Refactoring Phase

Notice that we have some recurring code.

```
import pytest
from Checkout import *

@pytest.fixture()
def checkout():
    checkout = Checkout()
    return checkout

def test_AddItemPrice(checkout):
    checkout.addItemPrice('a', 1)

def test_canAddITem(checkout):
    checkout.addItem("a")
```

Red Phase

Calculate Total

```
def test_canCalculateTotal(checkout):
   checkout.addItemPrice("a", 1)
   checkout.addItem("a")
   assert checkout.calculateTotal() == 1
```

Green Phase

```
class Checkout:
   def addItemPrice(self, item, price):
       pass
   def addItem(self, item):
       pass
   def calculateTotal(self):
       return 1
```

The return is hardcoded

Refactoring Phase

Remove `test_AddItemPrice(checkout) and test_canAddITem(checkout)`

Add Multiple Items and Calculate Total

Red Phase

Now we want to add multiple items and get correct total

```
def test_getCorrectTotalWithMultipleItems(checkout):
   checkout.addItemPrice("a", 1)
   checkout.addItem("a")
   checkout.addItemPrice("b", 2)
```

```
    checkout.addItem("b")
    assert checkout.calculateTotal() == 3
```

Green Phase

Let's fix the code

```
class Checkout:
    def __init__(self):
        self.prices = {}
        self.total = 0

    def addItemPrice(self, item, price):
        self.prices[item] = price

    def addItem(self, item):
        self.total += self.prices[item]

    def calculateTotal(self):
        return self.total
```

Add and Apply Discounts

Now we need to add a method to have a discount rule. It has 3 parameters - The item type the number of items required and the discount price

Red Phase

```
def test_addDiscountRule(checkout):
    checkout.addDiscount("a", 3, 2)
```

Green Phase

```
def addDiscountRule(self, item, nbrOfItems, price):
   pass
```

Refactoring Phase

```
import pytest
from Checkout import *

@pytest.fixture()
def checkout():
   checkout = Checkout()
   checkout.addItemPrice("a", 1)
   checkout.addItemPrice("b", 2)
   return checkout

def test_canCalculateTotal(checkout):
   checkout.addItem("a")
   assert checkout.calculateTotal() == 1

def test_getCorrectTotalWithMultipleItems(checkout):
   checkout.addItem("a")
   checkout.addItem("b")
   assert checkout.calculateTotal() == 3

def test_addDiscountRule(checkout):
   checkout.addDiscountRule("a", 3, 2)
```

For Apply Discount

Red Phase

```
def test_applyDiscountRule(checkout)
   checkout.addDiscountRule("a", 3, 2)
```

```
    checkout.addItem("a")
    checkout.addItem("a")
    checkout.addItem("a")
    assert checkout.calculateTotal() == 2
```

Green Phase

First lets add

```
@pytest.mark.skip
```

To the `test_applyDiscountRule(checkout)`

Right now we have a total that updates when an item is added

```
def addItem(self, item):
    self.total += self.prices[item]
```

But we don't know what discounts to apply until all the items are added so we want to move the calculation of the total to the `calculateTotal(self)` method.

Lets add a Discount class with the discount and items as variables and a dictionary keyed with items

Refactoring Phase

```
class Checkout:

    class Discount:
        def __init__(self, nbrOfItems, price):
            self.nbrOfItems = nbrOfItems
            self.price = price

    def __init__(self):
        self.prices = {}
        self.total = 0
        self.discount_dict = {}
        self.items = {}
```

```
def addItemPrice(self, item, price):
    self.prices[item] = price

def addItem(self, item):
    if item in self.items:
        self.items[item] += 1
    else:
        self.items[item] = 1

def calculateTotal(self):
    total = 0
    for item, cnt in self.items.items():
        total += self.prices[item] * cnt
    return total

def addDiscountRule(self, item, nbrOfItems, price):
    discount = self.Discount(nbrOfItems, price)
    self.discount_dict[item] = discount
```

Red Phase

Test still failing

Green Phase

```
def calculateTotal(self):
   total = 0
   for item, cnt in self.items.items():
       if item in self.discount_dict:
           discount = self.discount_dict[item]
           if cnt >= discount.nbrOfItems:
               nbrOfDiscounts = cnt/discount.nbrOfItems
               total += nbrOfDiscounts * discount.price
               remaining = cnt % discount.nbrOfItems
               total += remaining * self.prices[item]
```

```
            else:
                total += self.prices[item] * cnt
        else:
            total += self.prices[item] * cnt
    return total
```

Refactoring Phase

```
class Checkout:

    class Discount:
        def __init__(self, nbrOfItems, price):
            self.nbrOfItems = nbrOfItems
            self.price = price

    def __init__(self):
        self.prices = {}
        self.total = 0
        self.discount_dict = {}
        self.items = {}

    def addItemPrice(self, item, price):
        self.prices[item] = price

    def addItem(self, item):
        if item in self.items:
            self.items[item] += 1
        else:
            self.items[item] = 1

    def calculateTotal(self):
        total = 0
        for item, cnt in self.items.items():
            total += self.calculateItemTotal(item, cnt)
        return total

    def calculateItemTotal(self, item, cnt):
        total = 0
        if item in self.discount_dict:
            discount = self.discount_dict[item]
            if cnt >= discount.nbrOfItems:
                total += self.calculateItemDiscountTotal(item, cnt, discount)
            else:
                total += self.prices[item] * cnt
        else:
            total += self.prices[item] * cnt
        return total

    def calculateItemDiscountTotal(self, item, cnt, discount):
        total = 0
        nbrOfDiscounts = cnt / discount.nbrOfItems
        total += nbrOfDiscounts * discount.price
        remaining = cnt % discount.nbrOfItems
        total += remaining * self.prices[item]
        return total

    def addDiscountRule(self, item, nbrOfItems, price):
        discount = self.Discount(nbrOfItems, price)
        self.discount_dict[item] = discount
```

Throw Exception when Adding an Item with No Price

Red Phase

```
def test_ExceptionWithBadITem(checkout):
   with pytest.raises(Exception):
       checkout.addItem("c")
```

Green Phase

```
def addItem(self, item):
   if item not in self.prices:
       raise Exception("Bad Item")
```

```
    if item in self.items:
        self.items[item] += 1
    else:
        self.items[item] = 1
```

Refactoring Phase

–

Section 6: Test Doubles

Overview of Test Doubles, unitest.mock, and monkeypatch

Now we are going to go over how you can make sure we're running our unit test in isolation using the concepts of Dummies, Fakes, Stubs, Spies, and Mocks

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 oftentimes 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.

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 usable 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

- 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

- 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 provides the Mock class which is an extremely powerful class that can 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.

```
# Example
def test_Foo():
        bar = Mock()
        functionThatUsesBar( bar )
        bar.assert_called_once()
```

Mock Initialization

- 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.

```
# Example
def test_Foo():
        bar = Mock(spec=SpecClass)
        bar2= Mock(side_effect= barFunc)
        bar3 = Mock(return_value=1)
```

Mock Verification

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 - 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

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.

```
def callIt()
        print("Hello World")

def test_patch(monkeypatch)
        monkeypatch(callIt, Mock())
        callIt()
        callIt.assert_called_once()
```

unittest.mock Example

To Do

- Can call ReadFile
- readFromFile returns correct string
- readFromFile throws exception when file does

Red Phase

```
from LineReader import *
def test_canCallReadFile():
   readFromFile("file")
```

Green Phase

```
def readFromFile(filename):
   pass
```

Refactoring Phase

None

Red Phase

In the next test case, I want the function to actually do the work of opening a file, reading a line and returning it. I don't actually want to have to open a file for this test though, as that puts an external dependency on the test and potentially slows the test down.

Instead, I'll mock out the open function to return a magic mock object and I'll add a read line attribute to the mock, which is also a magic mock object that will return the test string. Then I'll call the read from file function, and when that call returns, I'll validate that the open function was called with the correct parameters and that the expected test string was returned.

```
def test_returnsCorrectString(monkeypatch):
   mock_file = MagicMock()
   mock_file.readline = MagicMock(return_value="test line")
   mock_open = MagicMock(return_value=mock_file)
   monkeypatch.setattr("builtins.open", mock_open)
   result = readFromFile("blah")
   mock_open.assert_called_once_with("blah", "r")
   assert result == "test line"
```

Green Phase

```
def readFromFile(filename):
   infile = open(filename, "r")
   line = infile.readline()
   return line
```

Refactoring Phase

Remove `test_canCallReadFile().`

It is redundant

Red Phase

I'm going to assume the production code will use the os.path exist function to verify the file exists and then throw an exception if it doesn't.
I'll mock out the os.path exist function so I can control when it returns. True or false depending on the test case without actually having to make modifications in the file system.

```
def test_throwsExceptionWithBadFile(monkeypatch):
   mock_file = MagicMock()
   mock_file.readline = MagicMock(return_value="test line")
   mock_open = MagicMock(return_value=mock_file)
   monkeypatch.setattr("builtins.open", mock_open)
   mock_exists = MagicMock(return_value = False)
   monkeypatch.setattr("os.path.exists", mock_exists)
   with raises(Exception):
       result = readFromFile("Blah")
```

Green Phase

Now that I have a failing unit test, I'll modify the production code to make it pass. I'm going to update the read from file function to use the Os.path exist method to verify the passed in file name exists and then it doesn't raise an exception.

```
def readFromFile(filename):
   if not os.path.exists(filename):
       raise Exception("Bad File")
   infile = open(filename, "r")
   line = infile.readline()
   return line
```

Refactoring Phase

In this test case, I passed in a fake file name and path.
So now the production code is actually generating an exception because the real os.path exist function is checking to see if that file exists and it's returning false. I need to update the test case to mock out if the os.path exists and have it return true for that particular test case.

```
def test_returnsCorrectString(monkeypatch):
   mock_file = MagicMock()
   mock_file.readline = MagicMock(return_value="test line")
   mock_open = MagicMock(return_value=mock_file)
   monkeypatch.setattr("builtins.open", mock_open)
   mock_exists = MagicMock(return_value=False)
   monkeypatch.setattr("os.path.exists", mock_exists)
   result = readFromFile("blah")
   mock_open.assert_called_once_with("blah", "r")
   assert result == "test line"
```

Now both of my tests are passing and I can enter the refactoring phase. There's quite a bit of duplication between the two tests. I'm going to create a test fixture for creating the first setup of mocks that are common between the two tests.

Section 7: TDD Best Practices

Always Do the Next Simplest Test Case

- This allows you to gradually increase the complexity of the code, refactoring as you go. This helps keep your code clean and understandable.
- If you jump to the complex cases too quickly you can find yourself stuck writing a lot of code for one test case which breaks the short feedback cycle we look for with TDD.
- Beyond just slowing you down this can also lead to bad design as you can miss some simple implementations that come from the incremental approach.

Use Descriptive Test Names

- The code is read 1000’s of times more than it’s written as the years go by. Making the code clear and understandable should be the top priority.
- Unit tests are the best documentation for the developers that come after you for how you intended your code to work. If they can’t understand what the unit test is testing that documentation value is lost.
- Test suites should name the class or function that is under test and the test name should describe the functionality that is being tested.

Keep Test Fast

Keep your unit tests building and running fast.

- One of the biggest benefits of TDD is the fast feedback on how your changes have affected things.
- You lose this if the build and/or execution of your unit tests is taking a long time (i.e. more than a few seconds).
- To help your tests stay fast try to:
  - Keep console output to a minimum (or eliminate it all together). This output just slows down the test and clutters up the test results.

- Mock out any slow collaborators that are being used with test doubles that are fast.

Use Code Coverage Analysis Tools

- Once you've implemented all your test cases go back and run your unit tests through a code coverage tool.
- It can be surprising some of the areas of your code you'll miss (especially negative test cases).
- You should have a goal of 100% code coverage on functions with real logic. Don't waste your time on one line getter and setter functions.

Run your unit tests multiple times and in a random order

- Running your tests many times will help ensure that you don't have any flaky tests that are failing intermittently.
- Running your tests in random order ensures that your tests don't have dependencies between each other.
- You can use the pytest-random-order plugin to randomize the execution of the tests and pytest-repeat for repeating all or a sub-set of the unit tests as needed.

Use a Static Code Analysis Tool

Using a static code analysis tool regularly on your code base is another core requirement for ensuring code quality.

- Pylint is an excellent open source static analysis tool for python that can be used for detecting bugs in your code.
- It can also verify the code is formatted to the team's standards - And it can even generate UML diagrams based on it's analysis
- In the last lecture I'll review what was gone over in the course and where do go from here