Module 5 – Unit Testing and TDD

Unit Testing in Python

Topics

- Types of Testing
- Unit Testing Vocabulary
- Test Case Design
- Testing Functions
- Testing Classes/Objects

Unit Testing Fundamentals

A Unit is a Small Piece of Code

A method or function

A module or class

A small group of related classes

An Automated Unit Test

Is designed by a human

Runs without intervention

Reports either ‘pass’ or ‘fail’

Strictly Speaking..

It’s not a unit test if it uses...

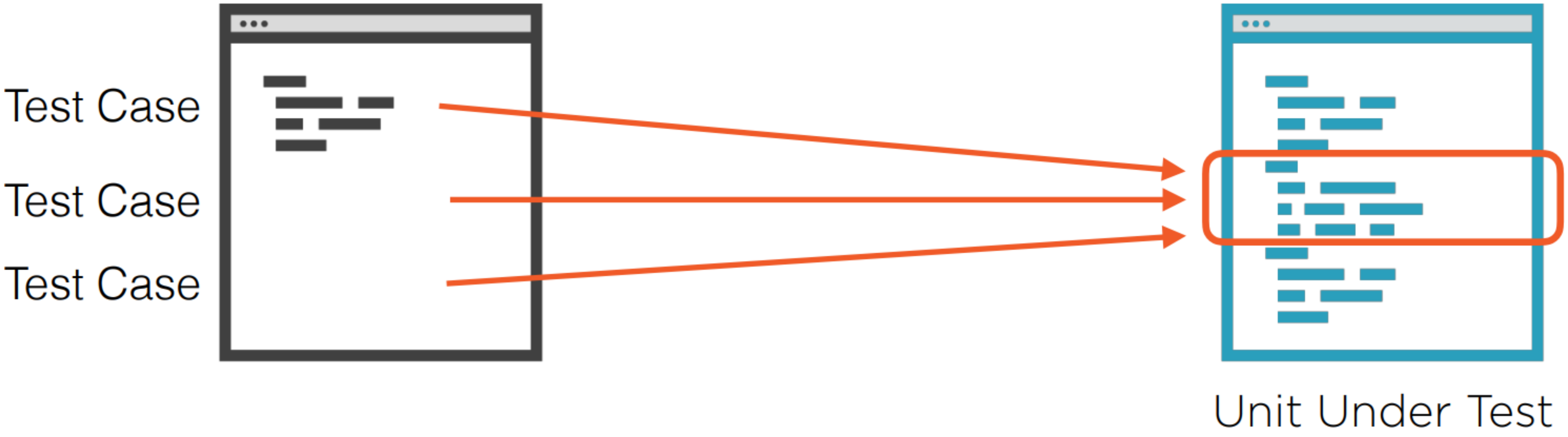
the Filesystem

a Database

the Network

(But it might still be a useful test)

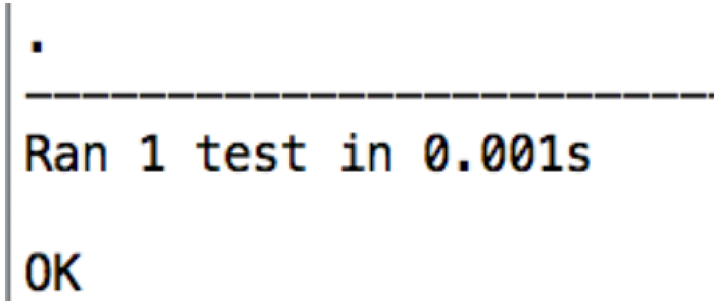
Unit Test Vocabulary: Test Case



Unit Test Vocabulary: Test Runner

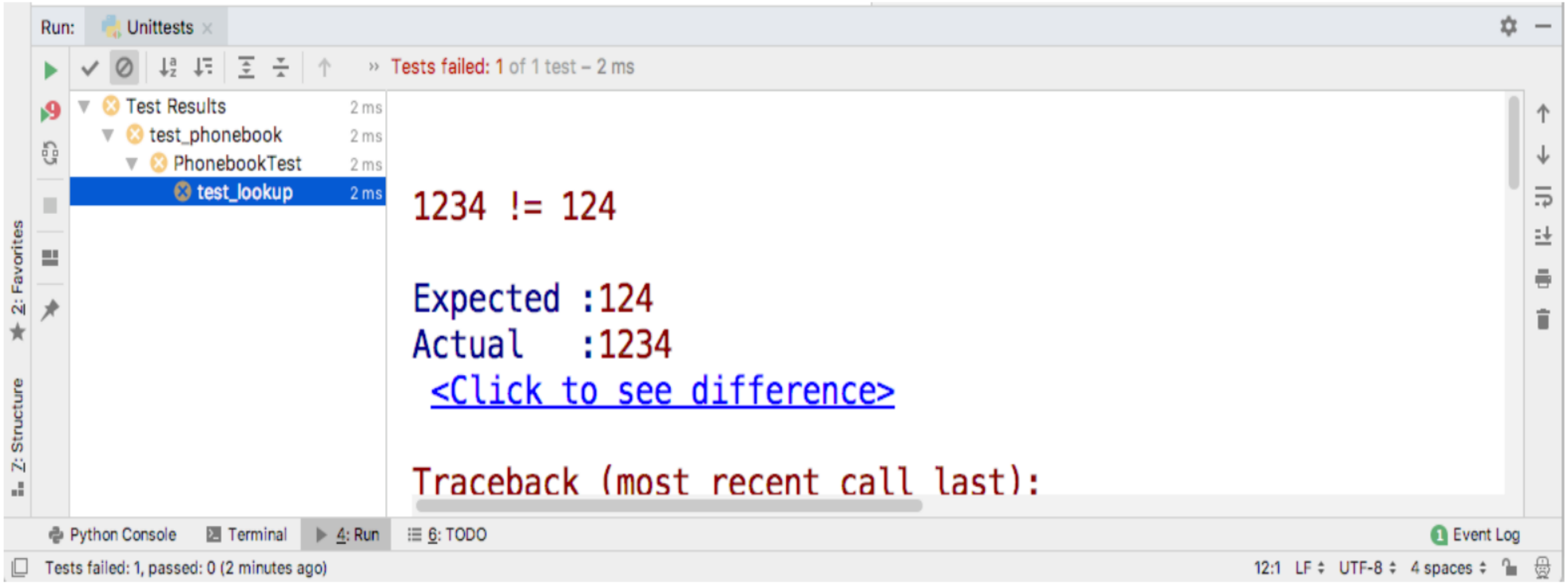
Test Case

Unit Under Test



```
.
------------------------
Ran 1 test in 0.001s

OK
```

Test Runner



```
1234 != 124

Expected :124
Actual   :1234
 <Click to see difference>

Traceback (most recent call last):
```

Tests failed: 1, passed: 0 (2 minutes ago)

Test Runner in PyCharm

Choosing a Test Runner

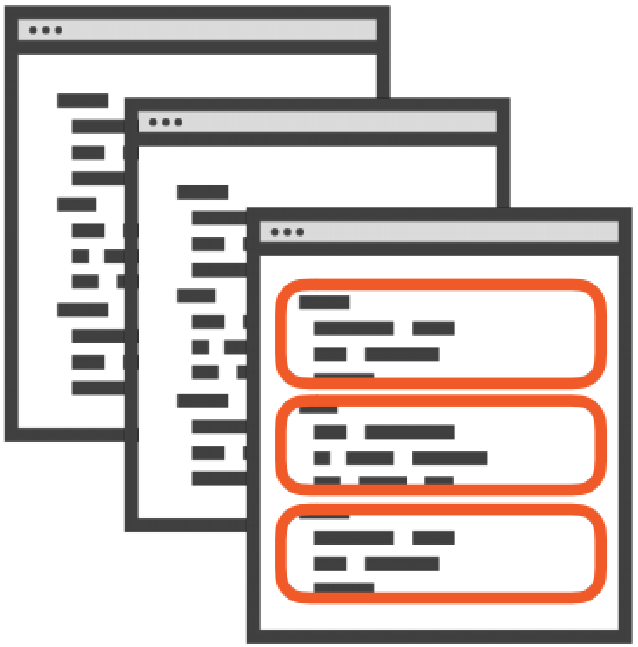
Working Interactively

An IDE like PyCharm

Continuous Integration

A Command Line Test Runner

Unit Test Vocabulary

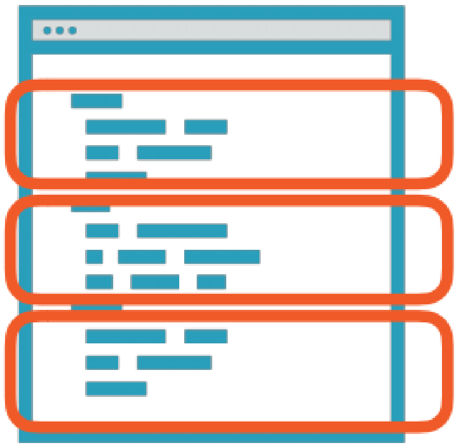


Test Case

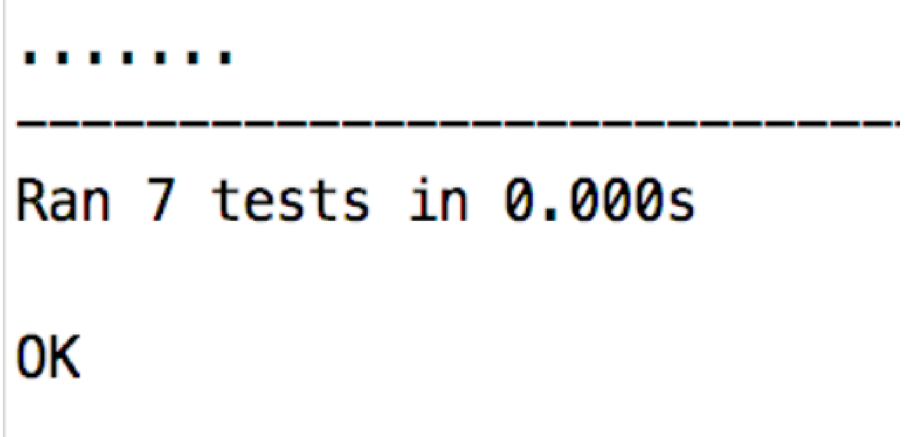
Test Case

Test Case

Test Suite



Units Under Test



```
.......
----------------------------------------------------------------------
Ran 7 tests in 0.000s

OK
```

Test Runner

Test Fixture: Order of Execution

setUp()

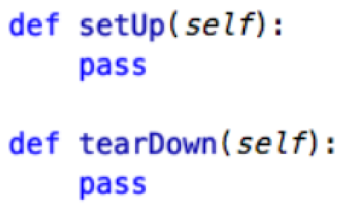
TestCaseMethod()

tearDown()

Unit Test Vocabulary

Test Case

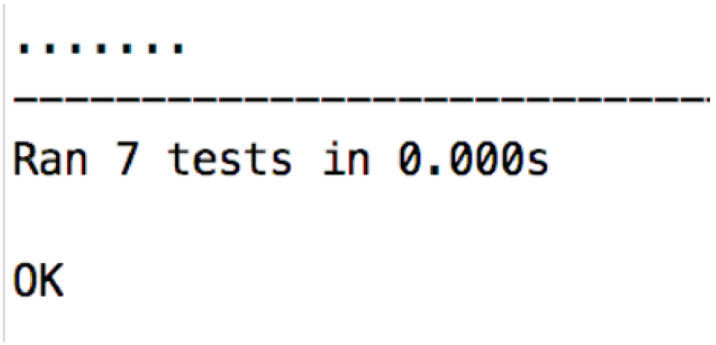
Test Suite



```
def setUp(self):
    pass

def tearDown(self):
    pass
```

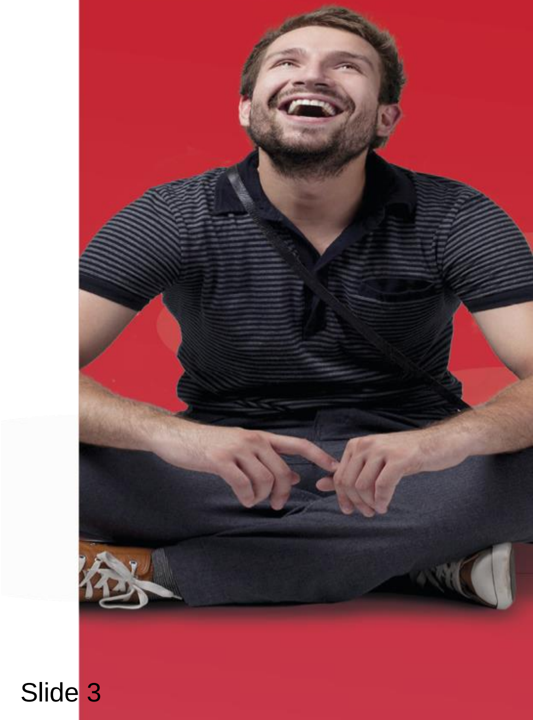
Test Fixture

Unit Under Test



```
.......
----------------------------------------------------------------------
Ran 7 tests in 0.000s

OK
```

Test Runner



The Three Parts of a Test

Arrange

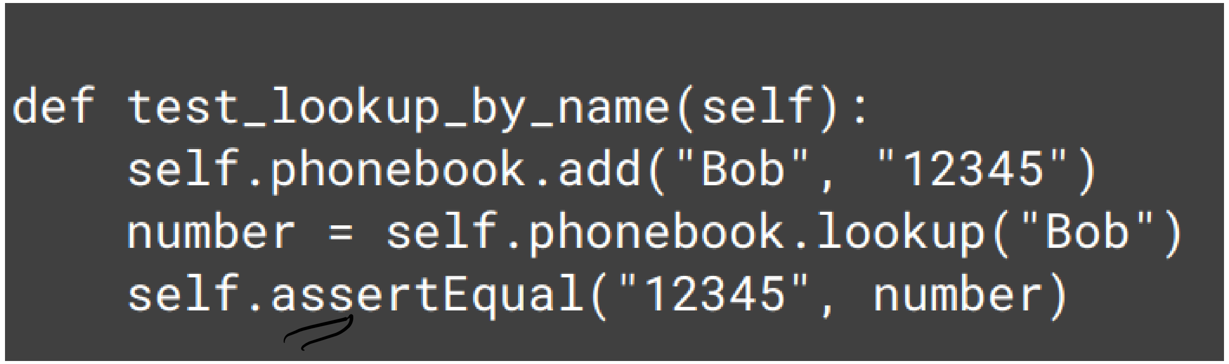
Set up the object to be tested, and collaborators

Act

Exercise the unit under test

Assert

Make claims about what happened

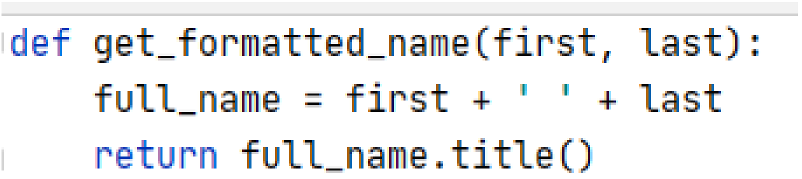


```
def test_lookup_by_name(self):
    self.phonebook.add("Bob", "12345")
    number = self.phonebook.lookup("Bob")
    self.assertEqual("12345", number)
```

- Test Case Name
- Arrange
- Act
- Assert

Testing a Function

- To learn about testing, we need code to test. Here's a simple function that takes in a first and last name, and returns a neatly formatted full name:



```
def get_formatted_name(first, last):
    full_name = first + ' ' + last
    return full_name.title()
```

- The function **get_formatted_name()** combines the first and last name with a space in between to complete a full name, and then capitalises and returns it.

- So when we call the function as shown, we get the following output:

```
print("Full name: " + get_formatted_name(peter, parker))
```

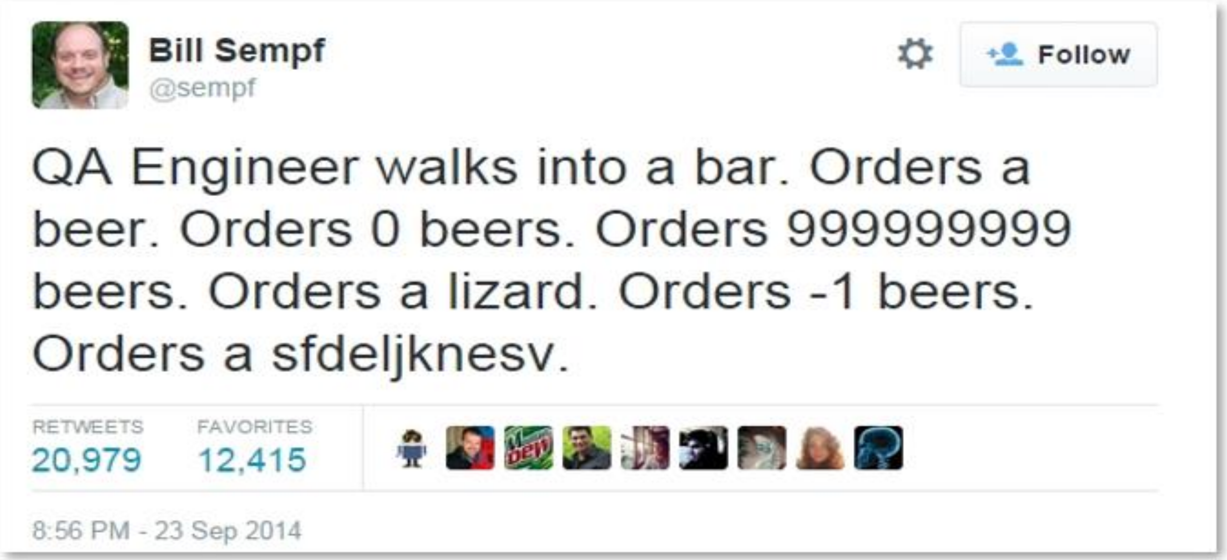
Prints: "Full name: Peter Parker"

```
print( "Full name: " + get_formatted_name(bob, dylan))
```

Prints: "Full name: Bob Dylan"

Testing a Function (Cont'd)

- We can see from using the function that it works correctly, but let's say that we wanted to modify the function so it can handle middle names as well.
- When doing this, we want to make sure that we don't break the functionality to provide a full name that only consists of a first and last name…
- …and to do *that* we're going to write a few **unit tests** for the function that can automatically determine if the function is working as planned!



Bill Sempf
@sempf

Follow

QA Engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 999999999 beers. Orders a lizard. Orders -1 beers. Orders a sfdeljknesv.

RETWEETS 20,979 FAVORITES 12,415

8:56 PM - 23 Sep 2014

A Passing Test

- The syntax for setting up a test case takes some getting used to, but once you've set up the test case it's straightforward to add more unit tests for your functions.
- To write a test case for a function, start by importing the `unittest` module and the function you want to test.
- Then create a class that inherits from `unittest.TestCase`, and write a series of methods to test different aspects of your function's behaviour.
- On the following slide there's a test case with one method that verifies that the function **get_formatted_name()** works correctly when given a first and last name.

Do you remember the difference between a **unit test** and a **test case**?

A Passing Test (Cont’d)

```
import unittest
from name_function import get_formatted_name

class NamesTestCase(unittest.TestCase):
    """Tests for 'name_function.py'."""

    def test_first_last_name(self):
        """Do names like 'Bob Dylan' work?"""
        formatted_name = get_formatted_name('bob', 'dylan')
        self.assertEqual(formatted_name, "Bob Dylan")

unittest.main()
```

Class inherits (i.e. is a subclass of) the **unittest.TestCase** class.

- First we import **unittest** and the function we want to test **get_formatted_name**, then we create a class called **NamesTestCase**, then add a unit test to it called **test_first_last_name**.

A Passing Test (Cont’d)

- When we then run our **test_name_function.py** program, we get the following output:

```
.
-------------------------------------------------------
Ran 1 test in 0.000s
OK
```

- The **dot** on the first line of output tells us that a single test passed.

- The next line tells us that Python ran **one** test, and it took **less than 0.001 seconds** to run.

- The final **OK** tells us that all unit tests in the test case passed.

Calculator.py Demo

Testing a Class

- We've just proved that we can write unit tests for a **function**, so now we'll move on to writing tests for a **class**.

- You'll use classes in many of your own programs, so it's helpful to be able to prove that your classes work correctly – and just like before, if you have passing tests for a class you're working on, you can be confident that improvements you make to the class won't accidentally break its current behaviour.

- Python provides a number of assert methods in the **unittest.TestCase** class. As mentioned earlier, ***assert*** methods test whether a condition you believe is true at a specific point in your code is indeed true.

- If the condition is true as expected, your assumption about how that part of your program behaves is confirmed & you can be confident that no errors exist…

- …while if the condition you assume is true is actually not true, then Python raises an exception so you know there's a problem and can deal with it.

Assert Methods

- Let's try out each of the assert methods - starting with the ones that check for equality or inequality:

```
assertEqual(3, 3)     # True - test passes.
assertEqual(3, 4)     # False - test fails.

assertNotEqual(3, 4) # True - test passes.
assertNotEqual(3, 3) # False - test fails.

assertTrue(3 == 3)    # True - test passes.
assertTrue(3 == 4)    # False - test fails.

assertFalse(3 == 4)   # True - test passes (3 is not equal to 4)
assertFalse(3 == 3)   # False - test fails (3 is not equal to 3)
```

Assert Methods (Cont'd)

- Next let's take a look at examples of the assertions dealing with lists…

```
my_list = ['milk', 'bread', 'cheese']
item = 'bread'

assertIn(item, my_list)          # True - item is in the list
assertIn('carrots', my_list)     # False - item is not in the list
```

- The **assertNotIn** functions work just like you'd expect them to:

```
assertNotIn('carrots', my_list)   # True - item is not in list
assertNotIn('bread', my_list)     # False - item is in list
```

Assert Methods (Cont'd)

- The final assertion we'll look at is whether a function raises an exception when given specific data to work with. For example:

```
def square_value(some_number):
    if ( str(some_number).isdigit() == false):
        raise Exception('Value must be of a numerical type!')
    else:
        return some_number * some_number
```

- So our function will square and return a value if we give it a number to work with, otherwise it will raise an exception. We can test for this via assertion like this:

```
# True - exception raised because 'three' is not a number.
assertRaises(Exception, square_value, 'three')

# False - no exception raised because 3 actually is a number.
assertRaises(Exception, square_value, 3)
```

Reminder: The **isdigit()** method checks to ensure that all characters in a string are numbers. If so it returns true, otherwise it returns false.

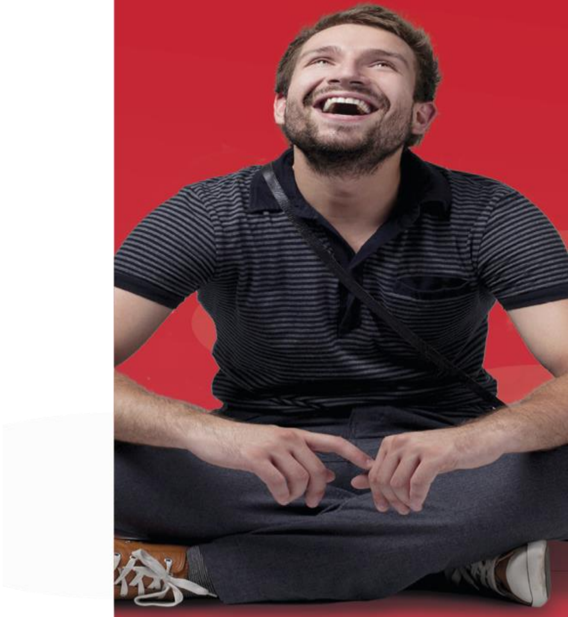
Responding to a Failed Test

- What do you do when a test fails?
- Assuming you’re checking the right conditions, a passing test means the function is behaving correctly and a failing test means there’s an error in the new code you wrote.
- So when a test fails, **don’t change the test**! Instead, **fix the code** that caused the test to fail by examining the changes you just made to the function, and figure out how those changes broke the desired behaviour!

A Class to Test

- Testing a class is similar to testing a function, but there are a few minor differences. Let's write a simple class to test that helps administer anonymous surveys:

```
class AnonymousSurvey():
    """Collect anonymous answers to a survey question."""

    def __init__(self, question):
        """Store a question, and prepare to store responses."""
        self.question = question
        self.responses = []

    def show_question(self):
        """Show the survey question."""
        print(question)

    def store_response(self, new_response):
        """Store a single response to the survey."""
        self.responses.append(new_response)

    def show_results(self):
        """Show all the responses that have been given."""
        print("Survey results:")
        for response in responses:
            print('- ' + response)
```



A Class to Test (Cont’d)

- Now let's try creating and using an instance of the class:

```
from survey import AnonymousSurvey

# Define a question, and make a survey.
question = "What language did you first learn to speak?"
my_survey = AnonymousSurvey(question)

# Show the question, and store responses to the question.
my_survey.show_question()
print("Enter 'q' at any time to quit.\n")
while True:
    response = input("Language: ")
    if response == 'q':
        break
    my_survey.store_response(response)

# Show the survey results.
print("\nThank you to everyone who participated in the survey!")
my_survey.show_results()
```

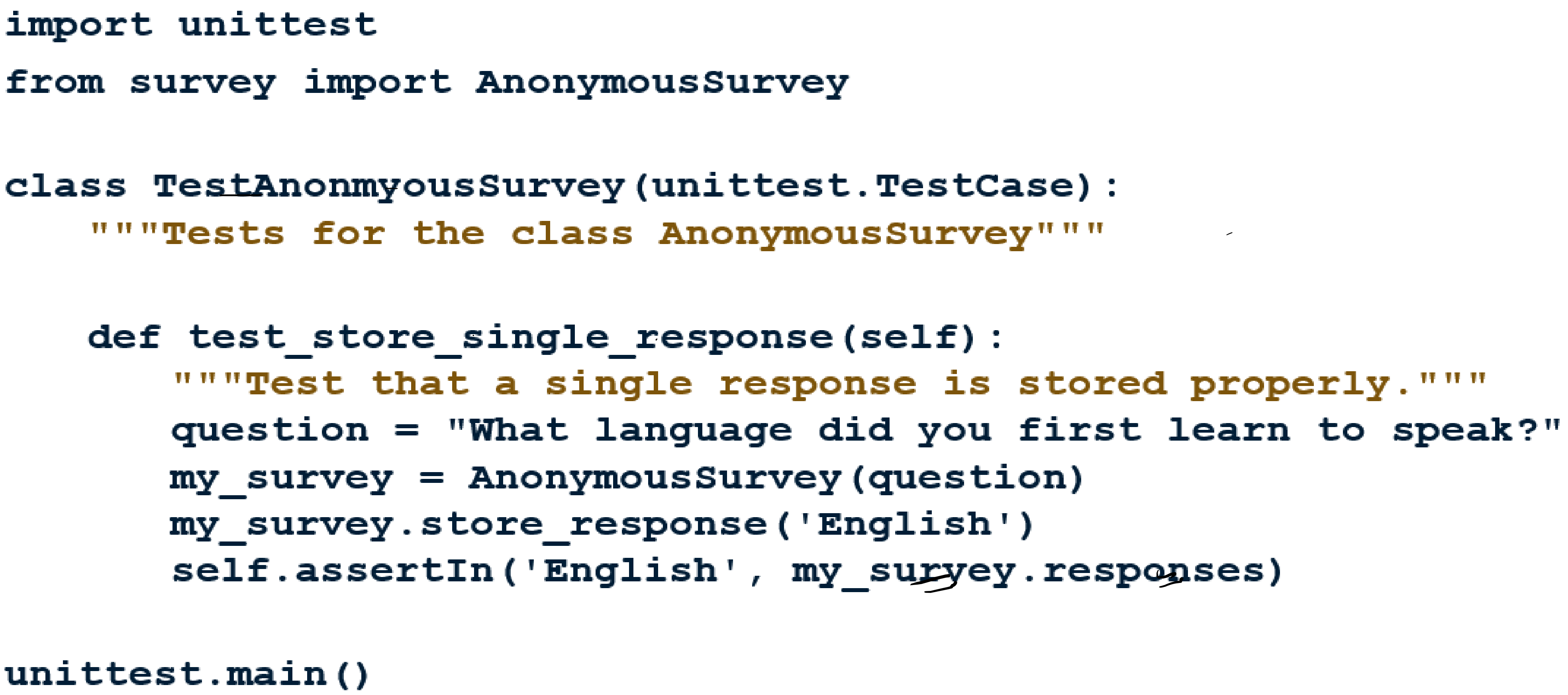
A Class to Test (Cont’d)

- So an example run of the program may look something like this:

```
What language did you first learn to speak?
Enter 'q' at any time to quit.

Language: English
Language: Spanish
Language: English
Language: Mandarin
Language: q

Thank you to everyone who participated in the survey!
Survey results:
- English
- Spanish
- English
- Mandarin
```

Testing the AnonymousSurvey Class

- Let’s write a test that verifies one aspect of the way **AnonymousSurvey** behaves. We’ll write a test to verify that a single response to the survey question is stored properly, by using the **assertIn()** method to verify that the response is in the list of responses after it’s been stored:



```
import unittest
from survey import AnonymousSurvey

class TestAnonmyousSurvey(unittest.TestCase):
    """Tests for the class AnonymousSurvey"""

    def test_store_single_response(self):
        """Test that a single response is stored properly."""
        question = "What language did you first learn to speak?"
        my_survey = AnonymousSurvey(question)
        my_survey.store_response('English')
        self.assertIn('English', my_survey.responses)

unittest.main()
```

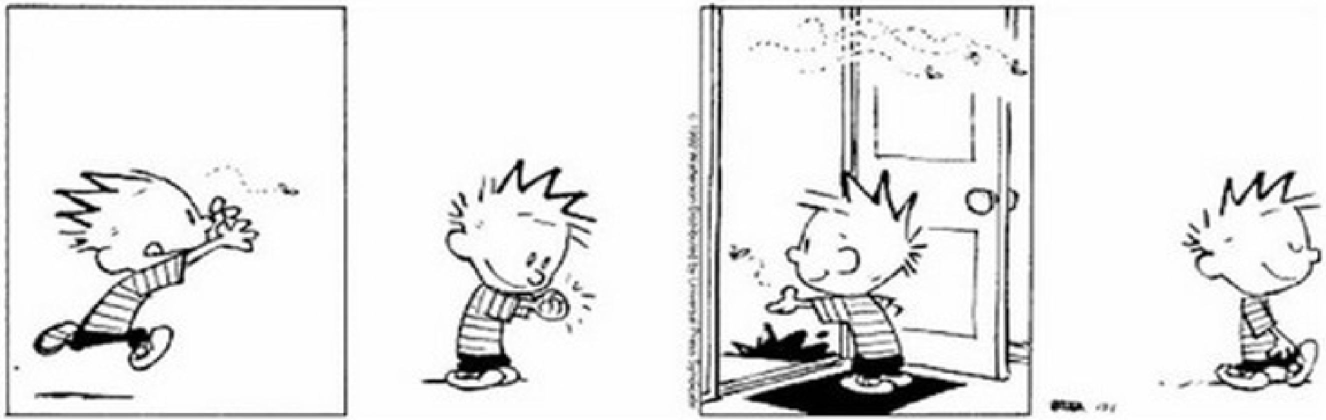
Testing the Anonymous Survey Class (Cont’d)

- When we run **test_survey.py**, the test passes and shows the following output:

```
.
-------------------------------------------------------
Ran 1 test in 0.000s
OK
```

- This is good, but a survey is only useful if it generates more than one response – so let's verify that three responses can be stored correctly by adding the following test to our **survey_test.py** test case…

Regression: When you fix one bug but you introduce others!

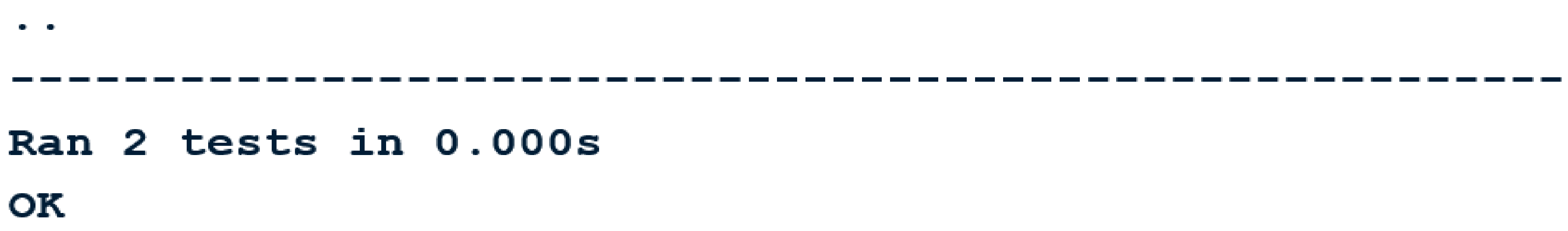


Testing the Anonymous Class (Cont’d)

```python
import unittest
from survey import AnonymousSurvey

class TestAnonmyousSurvey(unittest.TestCase):
    """Tests for the class AnonymousSurvey"""

    def test_store_single_response(self):
        """Test that a single response is stored properly."""
        # Previous code here...

    def test_store_three_responses(self):
        """Test that three responses are stored properly"""
        question = "What language did you first learn to speak?"
        my_survey = AnonymousSurvey(question)
        responses = ['English', 'Spanish', 'Mandarin']

        for response in responses:
            my_survey.store_response(response)

        for response in responses:
            self.assertIn(response, my_survey.responses)


unittest.main()
```

Testing the Anonymous Class (Cont’d)

- Now when we run **test_survey.py**, the tests pass and show the following output:



```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s
OK
```

- This works perfectly – however, the tests are a bit repetitive, so we'll use another feature of **unittest** to make them more efficient.

- The **unittest.TestCase** class has a **setUp()** method that allows you to create objects once, and then use them in each of your test methods (so you don't need to create new objects to test in each individual test!).

- When you include a **setUp()** method in a **TestCase** class, Python runs that method **before** running any methods that start with the name **test_** - so let's modify our test case to use that functionality.

Testing the Anonymous Class (Cont’d)

```python
import unittest
from survey import AnonymousSurvey

class TestAnonymousSurvey(unittest.TestCase):
    """Tests for the class AnonymousSurvey."""

    def setUp(self):
        """Create a survey & responses for use in all test methods."""
        question = "What language did you first learn to speak?"
        self.my_survey = AnonymousSurvey(question)
        self.responses = ['English', 'Spanish', 'Mandarin']

    def test_store_single_response(self):
        """Test that a single response is stored properly."""
        self.my_survey.store_response(self.responses[0])
        self.assertIn(self.responses[0], self.my_survey.responses)

    def test_store_three_responses(self):
        """Test that three individual responses are stored properly."""
        for response in self.responses:
            self.my_survey.store_response(response)
        for response in self.responses:
            self.assertIn(response, self.my_survey.responses)

unittest.main()
```

Testing Wrap-UP

- At this point we've learned to write tests for functions and classes using tools in the **unittest** module, how to write a class that inherits from **unittest.TestCase**, and how to write test methods that verify specific behaviours that our functions and classes should exhibit.

- We've also learned to use the **setUp()** method to efficiently create instances and attributes of our classes that can be used in ***all*** the **test_** methods or our class.

- Testing is an important topic that many beginners don’t learn. Just remember that you don’t have to write tests for all the simple projects you try as a beginner, but as soon as you start to work on projects that involve significant development effort, you should test the critical behaviours of your functions and classes.

- By doing so, you’ll be more confident that new work on your project won’t break the parts that work, and this will give you the freedom to make improvements to your code.

- If you accidentally break existing functionality, you’ll know right away, so you can still fix the problem easily – because responding to a failed test that you ran is much easier than responding to a bug report from an unhappy user!

Module 5 – Lab Activities