Lesson 36. Testing and debugging your programs

After reading lesson 36, you'll be able to

- Use the unittest library
- Write tests for your program
- Efficiently debug your programs

It's unlikely that you'll write a perfect program on the first try. Often you'll write code, test it with a few inputs, make changes to it, and test it again, repeating this process until your program runs as expected.

Consider this

Think about your experience in programming so far. When you write a program and it doesn't work, what kinds of things do you do to fix it?

Answer:

Look at the error message, if any, and see whether there are clues like line numbers that can direct me toward the issue. Put `print` statements at certain locations. Try a different input.

36.1. WORKING WITH THE UNITTEST LIBRARY

Python has many libraries that can help you create a testing structure around your program. A testing library is especially useful when you have programs that contain different functions. One testing library comes with your Python installation, and its documentation for Python 3.5 is available at https://docs.python.org/3.5/library/unittest.html.

To create a suite of tests, you create a class representing that suite. Methods inside that class represent different tests. Every test that you want to run should have `test_` as a prefix, and then any name for the method. The following listing contains code that defines two simple tests and then runs them.

Listing 36.1. Simple test suite

```
import unittest                                    1

class TestMyCode(unittest.TestCase):               2
    def test_addition_2_2(self):                   3
        self.assertEqual(2+2, 4)                   4
    def test_subtraction_2_2(self):                5
        self.assertNotEqual(2-2, 4)                6

unittest.main()                                    7
```

- **1 Imports the unittest library**
- **2 Class for the suite of tests to do**
- **3 Method to test whether 2 + 2 is 4**
- **4 Test checks whether 2 + 2 is 4 by asserting that the 2 + 2 and 4 are equal**
- **5 Method to test whether 2 - 2 isn't 4**
- **6 Test checks whether 2 - 2 isn't 4 by asserting that the 2 - 2 and 4 aren't equal**
- **7 Runs the tests defined in your test suite class**

The code prints the following:

```
Ran 2 tests in 0.001s
```

This is expected because the first test checks whether 2 + 2 is equal to 4, which is true. And the second test checks whether 2 - 2 isn't equal to 4, which is true. Now suppose that one of the tests is `False` by making the following change:

```
def test_addition_2_2(self):
        self.assertEqual(2+2, 5)
```

Now the test checks whether 2 + 2 is equal to 5, which is `False`. Running the test program again now prints the following:

```
FAIL: test_addition_2_2 (__main__.TestMyCode)
-----------------------------------------------------
Traceback (most recent call last):
  File "C:/Users/Ana/.spyder-py3/temp.py", line 5, in
    self.assertEqual(2+2, 5)
AssertionError: 4 != 5

-----------------------------------------------------
Ran 2 tests in 0.002s

FAILED (failures=1)
```

This message is rife with information. It tells you the following:

- Which test suite failed: `TestMyCode`
- Which test failed: `test_addition_2_2`
- Which line inside the test failed: `self.assertEqual(2+2, 5)`
- Why it failed, by comparing what value it got and what value it expected: `4 != 5`

Comparing values in this way is a bit silly. Clearly, you'd never have to check that 2 + 2 is 4. You usually need to test code that has more substance to it; typically, you want to make sure that functions do the correct thing. You'll see how to do this in the next section.

Quick check 36.1

Q1:

Fill in each of the following lines:

```
class TestMyCode(unittest.TestCase):
    def test_addition_5_5(self):
        # fill this in to test 5+5
    def test_remainder_6_2(self):
        # fill this in to test the remainder when
```

36.2. SEPARATING THE PROGRAM FROM THE TESTS

You should decouple the code you write as part of your program from the code that you write to test the program. Decoupling reinforces the idea of modularity, in that you separate code into different files. This way, you avoid cluttering the program itself with unnecessary testing commands.

Suppose you have one file that contains the two functions shown in listing 36.2. One function checks whether a number is prime (divisible by only 1 and itself, and not equal to 1) and returns `True` or `False`. The other returns the absolute value of a number. Each of these functions has an error in its implementation. Before reading further on how to write tests, can you spot the errors?

Listing 36.2. File, named funcs.py, containing functions to test

```
def is_prime(n):
    prime = True
    for i in range(1,n):
        if n%i == 0:
            prime = False
    return prime

def absolute_value(n):
    if n < 0:
        return -n
    elif n > 0:
        return n
```

In a separate file, you can write unit tests to check whether the functions making sure to try a variety of inputs. Creating tests for every function you write is called *unit testing* because you're testing each function's behavior by itself.

Definition

A *unit test* is a series of tests that checks whether the actual output matches the expected output for a function.

A common way of writing unit tests for a method is to use the Arrange Act Assert pattern:

- *Arrange*—Sets up objects and their values to pass to the function undergoing unit testing
- *Act*—Calls the function with the parameters set up previously
- *Assert*—Makes sure the function behavior is what was expected

The following listing shows the code for unit testing the functions in funcs.py. The class `TestPrime` contains tests related to the `is_prime` function, and the class `TestAbs` contains tests related to the `absolute_value` function.

Listing 36.3. File, named test.py, containing the tests

```
import unittest                                          1
import funcs                                             2

class TestPrime(unittest.TestCase):                      3
    def test_prime_5(self):                              4
        isprime = funcs.is_prime(5)                      5
        self.assertEqual(isprime, True)                  6
    def test_prime_4(self):
        isprime = funcs.is_prime(4)
        self.assertEqual(isprime, False)
    def test_prime_10000(self):
        isprime = funcs.is_prime(10000)
        self.assertEqual(isprime, False)

class TestAbs(unittest.TestCase):
    def test_abs_5(self):
        absolute = funcs.absolute_value(5)
        self.assertEqual(absolute, 5)
    def test_abs_neg5(self):
        absolute = funcs.absolute_value(-5)
        self.assertEqual(absolute, 5)
    def test_abs_0(self):
        absolute = funcs.absolute_value(0)
        self.assertEqual(absolute, 0)

unittest.main()
```

- **1 Brings in unittest classes and functions**
- **2 Brings in the functions you defined in funcs.py**
- **3 One class for a set of tests**
- **4 One test, with the method name describing what the test is for**
- **5 Calls the function is_prime from the funcs.py file with parameter 5 and assigns the return to isprime**
- **6 Adds the test that checks whether the result from the function call is True**

Running this code shows that it ran six tests and found two errors:

```
FAIL: test_abs_0 (__main__.TestAbs)
-------------------------------------------------------
Traceback (most recent call last):
  File "C:/Users/Ana/test.py", line 24, in test_abs_0
    self.assertEqual(absolute, 0)
AssertionError: None != 0

=======================================================
FAIL: test_prime_5 (__main__.TestPrime)
-------------------------------------------------------
Traceback (most recent call last):
  File "C:/Users/Ana/test.py", line 7, in test_prime_
    self.assertEqual(isprime, True)
AssertionError: False != True

-------------------------------------------------------
Ran 6 tests in 0.000s
```

With this information, you can make changes to your functions in funcs.py to try to fix them. The information provided here tells you the tests that failed: `test_abs_0` and `test_prime_5`. You can now go back to your function and try to fix it. This process is called *debugging* and is discussed in the next section.

Thinking like a programmer

Using descriptive names for the methods is useful for providing quick, at-a-glance, information when tests fail. You can include the function name, the inputs, and possibly a oneor two-word description of what it's testing.

It's important to make your changes incrementally. Every time you make a change, you should run the tests again by running tests.py. You do this to make sure that any changes you make to fix one issue won't cause another issue to pop up.

Quick check 36.2

Q1:

Modify the code in listing 36.2 to fix the two errors. After each change, run tests.py to see whether you fixed the errors.

36.2.1. Types of tests

The unittest library has a variety of tests you can perform, not just to check whether one value is equal to another using `assertEqual`. The complete list is in the documentation for the library. Take a moment to browse through the list.

Quick check 36.3

Looking at the list of tests you can do, which would be most appropriate in the following situations?

1

To check that a value is `False`

2

To check that a value is in a list

3

To check that two dictionaries are equal

36.3. DEBUGGING YOUR CODE

The process of debugging code is somewhat an art form in that there's no specific formula for how to do it. The process begins after you have a set of tests that have failed. These tests offer a starting point for where to look in the code and under what specific conditions; you have a set of inputs you gave to a function, which gave you output that wasn't what you expected.

Often, a brute-force solution for debugging is most efficient; this means being systematic about looking at every line, and using a pen and paper to note values. Starting with the inputs that caused the test to fail, pretend you're the computer and execute each line. Write down what values are assigned to each variable, and ask yourself whether that's correct. As soon as you find an incorrect value being calculated from what you expect, you've likely found where an error is occurring. Now it's up to you to figure out why the error is occurring, using what you've learned.

As you're tracing through your program by going line by line, a common mistake is to assume that simple lines of code are correct, especially if you're debugging your own code. Be skeptical of every line. Pretend that you're explaining your code to someone who doesn't know anything about programming. This process is called *rubber ducky debugging*, which means you explain your code to a real (or imaginary) rubber duck, or any other inanimate object. This forces you to explain the code in plain English, not programming jargon, and gets you to tell your helper what each line of code is trying to accomplish.

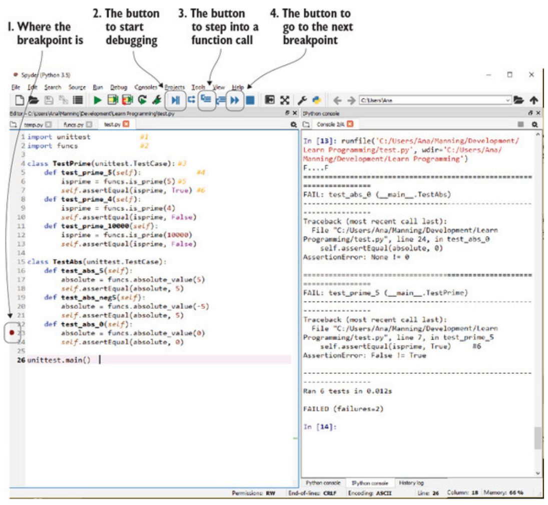
36.3.1. Using tools to help you step through code

Many tools have been designed to help make your debugging process easier step you take. It's still up to you to determine why a line of code is incorrect when the value for a variable isn't what you expect it to be. A debugger can be used on any code that you write.

You can use the Spyder debugger to identify the problem with the code shown in listing 36.2, using the test created in listing 36.3. You know that two tests failed: `test_abs_0` and `test_prime_5`. You should debug each one separately.

You'll now debug `test_abs_0`. Figure 36.1 shows what your Spyder editor should look like after you've opened tests.py. You have the editor on the left, the IPython console on the bottom right, and the variable explorer at the top right.

Figure 36.1. Screenshot of the debugging window


1. Where the breakpoint is
2. The button to start debugging
3. The button to step into a function call
4. The button to go to the next breakpoint
Spyder (Python 3.5)
File Edit Search Source Run Debug Consoles Projects Tools View Help
Editor - C:\Users\Ana\Manning\Development\Learn Programming\test.py
temp.py funcs.py test.py
IPython console
Console 2/A
Python console IPython console History log
Permissions: RW End-of-lines: CRLF Encoding: ASCII Line: 26 Column: 18 Memory: 66 %

```
1 import unittest                  #1
2 import funcs                     #2
3
4 class TestPrime(unittest.TestCase): #3
5     def test_prime_5(self):          #4
6         isprime = funcs.is_prime(5) #5
7         self.assertEqual(isprime, True) #6
8     def test_prime_4(self):
9         isprime = funcs.is_prime(4)
10        self.assertEqual(isprime, False)
11    def test_prime_10000(self):
12        isprime = funcs.is_prime(10000)
13        self.assertEqual(isprime, False)
14
15 class TestAbs(unittest.TestCase):
16    def test_abs_5(self):
17        absolute = funcs.absolute_value(5)
18        self.assertEqual(absolute, 5)
19    def test_abs_neg5(self):
20        absolute = funcs.absolute_value(-5)
21        self.assertEqual(absolute, 5)
22    def test_abs_0(self):
23        absolute = funcs.absolute_value(0)
24        self.assertEqual(absolute, 0)
25
26 unittest.main()
```

```
In [13]: runfile('C:/Users/Ana/Manning/Development/
Learn Programming/test.py', wdir='C:/Users/Ana/
Manning/Development/Learn Programming')
F....F
======================================================================
FAIL: test_abs_0 (__main__.TestAbs)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:/Users/Ana/Manning/Development/Learn
Programming/test.py", line 24, in test_abs_0
    self.assertEqual(absolute, 0)
AssertionError: None != 0

======================================================================
FAIL: test_prime_5 (__main__.TestPrime)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:/Users/Ana/Manning/Development/Learn
Programming/test.py", line 7, in test_prime_5
    self.assertEqual(isprime, True)     #6
AssertionError: False != True

----------------------------------------------------------------------
Ran 6 tests in 0.012s

FAILED (failures=2)

In [14]:
```

The first step is to put a breakpoint in your code (#1 in figure 36.1). A breakpoint indicates a spot in the code where you'd like to stop execution so that you can inspect the value. Because the test `test_abs_0` failed, put a breakpoint at the first line inside that method. You can insert a breakpoint by double-clicking the area just to the left on the line on which you want to put the breakpoint; a dot appears.

Then, you can start running the program in debugging mode. You do this by clicking the button with the blue arrow and two vertical lines (#2 in figure 36.1). The console now shows you the first few lines of the code and an arrow, ----->, indicating which line is to be executed:

```
----> 1 import unittest
      2 import funcs
      3
      4 class TestPrime(unittest.TestCase):
      5     def test_prime_5(self):
```

Click the blue double arrows (#4 in figure 36.1) to go to the breakpoint you put in. Now, the console shows where you are in the code:

```
     21         self.assertEqual(absolute, 5)
     22     def test_abs_0(self):
1--> 23         absolute = funcs.absolute_value(0)
     24         self.assertEqual(absolute, 0)
     25
```

You'd like to go into the function to see why the value for `absolute` isn't 0 and this test fails. Click the button with the small blue arrow that goes into three horizontal lines to "step into" the function (#3 in figure 36.1). Now you're inside the function call. Your variable explorer window should show you that the value for `n` is 0, and the console shows that you've just entered the function call:

```
      6     return prime
      7
----> 8 def absolute_value(n):
      9     if n < 0:
     10         return -n
```

If you click the Step Into button two more times, it takes you to the line

```
      9     if n < 0:
     10         return -n
---> 11     elif n > 0:
     12         return n
     13
```

At this point, you can see what's wrong. The `if` statement isn't executed because `n` is `0`. The `else` statement also isn't executed because `n` is `0`. You've likely figured out the issue and can now exit debugging by clicking the blue square. The function returns `None` because no case handles what the program should do when `n` is `0`.

Quick check 36.4

Use the debugger to debug the other test case that fails, `test_prime_5`:

1

Set a breakpoint at an appropriate line.

2

Run the debugger; go to the breakpoint.

3

Step into the function call and keep stepping through the program until you see the issue.

SUMMARY

In this lesson, my goal was to teach you the basics of testing and debugging and to give you more practice at importing and using libraries. I showed you how to use the unittest library to organize your tests and the importance of separating the code from the testing framework. You used a debugger to step through your code.

Let's see if you got this...

Q36.1

Here's a buggy program. Write unit tests and try to debug it:

```
def remove_buggy(L, e):
    """
    L, list
    e, any object
    Removes all e from L.
    """
    for i in L:
        if e == i:
            L.remove(i)
```

PREV
Lesson 35. Useful libraries

NEXT
Lesson 37. A library for graphical user interfaces