CSC108: Using unittest

Daniel Zingaro
University of Toronto
daniel.zingaro@utoronto.ca

November 2012

1 Revisiting Docstring Examples

So far, we've discussed including an example or two in our docstrings. Those examples serve two purposes:

- Giving the reader of `help` an example of how the function works, and
- Using `doctest` to automatically test these examples to make sure that the function works in those cases.

At the same time, we've also discussed providing comprehensive test suites, where we come up with categories of function calls and test the function with one example from each category.

For example, remember this function?

```
def num_vowels(s):
  '''(str) -> int
  Return number of vowels in s.

  >>> num_vowels('Europe')
  4
  '''
  counter  = 0
  for char in s:
    if char in 'aeiouAEIOU':
      counter = counter + 1
  return counter
```

To comprehensively test this function, we'd want to consider categories such as these:

- Empty string
- String of no vowels

- String of one vowel
- String of multiple vowels
- String of consonants and vowels
- String with nonalphabetic characters and vowels
- String with lowercase and uppercase vowels

We don't want to add tests for each of these categories to the docstring. This would make the docstring very long, and the purpose of docstrings is documentation (not thorough testing).

Instead, to write a test suite for one or more functions, we will use the `unittest` module. (Our docstrings will continue to include one or two examples, of course.)

2 Using unittest

`unittest` is a built-in Python module that automatically runs all tests in a test file. Carry out the following steps to use this module:

1. Create a new Python file that starts with the word `test`. For example, for the vowels example, you could call it `test_vowels.py`. In general, call it `test_x.py`, where `x` is the name of the module whose functions you are testing.
2. In your test file, start by importing the `unittest` module: `import unittest`.
3. Next, import the functions you want to test. This can take two forms
   - `import vowels`, or
   - `from vowels import num_vowels`

   The latter allows you to refer to `num_vowels` rather than `vowels.num_vowels`.
4. Now, for each function you want to test, create a new class named `Test_x`, where `x` is the name of the function you are going to test in this class. The class must inherit from `unittest.TestCase`.

Here's what we have so far in `test_vowels.py`:

```
import unittest
from vowels import num_vowels

class Test_num_vowels(unittest.TestCase):
```

5. Now, for each test case, create a new method inside of that class. Name the function `test_x`, where `x` is a description of the test case. For example, when testing the "empty string" category for `num_vowels`, name the method `test_empty_string`. The method should take no parameters, so use `(self)` as its parameter list.

6. Each method should make a call to one of the `assertXxx` methods that is available as a result of inheriting from `unittest.TestCase`. One such method is `assertEqual`, which takes three parameters: a first value, a second value, and a message to print if those values are not equal. Typically, the first value comes from a call to the function and the second is hard-coded. Here is a method that tests whether `num_vowels` works properly with the empty string.

```
  def test_empty_string(self):
    self.assertEqual(num_vowels(''), 0,
      'empty string')
```

7. Continue adding test-case methods to the class, one method per test case. Use `assertEqual` whenever you want to test that the function returns the expected value. You can also use `assertTrue` or `assertFalse` if the function returns a Boolean value. `assertTrue` and `assertFalse` take only two parameters: a value (returned from a call to the function) and a message to print if the function does not return `True` (for `assertTrue`) or `False` (for `assertFalse`).

8. If you are testing more than one function, do not place tests for multiple functions in the same class. Instead, create one class per function, and repeat the steps you followed above for creating a class and its methods.

9. At the bottom of your test module, add the following line:
   `unittest.main(exit=False)` Then, run your test module.

Here is a working test module for testing `num_vowels`. It contains only two tests, so it is not comprehensive (more tests are required, since we've only given examples for two of the `num_vowels` categories here).

```
import unittest
from vowels import num_vowels

class Test_num_vowels(unittest.TestCase):

  def test_empty_string(self):
    self.assertEqual(num_vowels(''), 0,
      'empty string')

  def test_one_vowel(self):
    self.assertEqual(num_vowels('e'), 1,
      'string of one vowel')

unittest.main(exit=False)
```

3 Exercise

Add the following function to `vowels.py`:

```
def any_vowels(s):
  '''(str) -> bool
  Return True iff s contains at least one vowel.
  '''

  return num_vowels(s) > 0
```

Add a new class to `test_vowels.py` that tests this function on two test cases.

4 Interpreting Test Results

Running `test_vowels.py` presents you with information on the tests that past and failed.

- The first line includes one character per test, so you should see a line of four characters as the first line of output. Each character will be a `.` (pass), `F` (fail), or `E` (error). Fail is different from error: fail means that your test failed to return the proper value, whereas error means that there is a problem running the test case. For example, if you forgot to add `any_vowels` to your `import` statement at the top of `test_vowels.py`, you'll see two `E`s on the first line of output.
- For any tests that produced an error or that failed, you'll see output telling you the name of the test method that failed, the output that the function produced in contrast to the expected output, and your message from the `assertXxx` method call.
- The bottom of the output tells you the total number of tests that were run and how many of those failed or errored.

5 Testing Functions that Return None

Recall this function:

```
def insert_after(L, n1, n2):
  '''(list of int, int, int) -> NoneType
  Insert n2 after each occurrence of n1 in L.
  '''
  i = 0
  while i < len(L):
    if L[i] == n1:
      L.insert(i+1, n2)
      i += 1
    i += 1
```

This function returns `None`. (It has no `return` statement, and the value returned automatically by such functions is `None`.) It is therefore incorrect to write a unittest test case that looks something like:

```
def test_empty_list(self): # incorrect
  self.assertEqual(insert_after([], 2, 3), [],
    'empty list')
```

The reason this is wrong is because the `insert_after` function returns `None`, whereas the `assertEqual` method is expecting a return value of `[]`. Therefore, this test will always fail.

Instead, what we have to do is set up a variable referring to a list, call `insert_after` on that list, then check that the variable refers to the modified list:

```
def test_empty_list(self): # correct
  lst = []
  insert_after(lst, 2, 3)
  self.assertEqual(lst, [],
    'empty list')
```

6 Exercise

Save the `insert_after` function in a file named `insert.py`. Then, in file `test_insert.py`, write two test cases for `insert_after`.