Anatomy of a Test File

CIS 192

January 30, 2017

1 Unit Testing in Python

Unit tests in Python, like in all other languages and programming situations, are useful tools to check your work and specify the behavior your code should implement. This document will help you write a test file with useful test cases.

2 The Anatomy of a Test File

2.1 Initial Setup

```
1  import unittest
2  from hw2 import *
```

The above image shows what a typical test file header will look like. The top line imports the unittest library, which is responsible for providing assertion tests that you use in this file. The next line allows you to import functions from the file you wish to text. Using the * allows you to automatically import all functions from your file, which in this case is "hw2.py".

2.2 The Tester Class

```
5  class TestHomework2(unittest.TestCase):
6
7      def setUp(self):
8          pass
```

You'll notice here that we define a class – this may be familiar to you from experience in Java/other languages. We'll cover information about classes by February 8, but until then you just need to understand that this class provides the framework for a testing file. The important thing to remember for now is that each method in the class must now take "self" as its first argument. More on that later.

The class is called "TestHomework2" here; yours can be anything that makes sense. It is important to give the "unittest.TestCase" argument to this class so that it demonstrates all of the behaviors we want. This class will now run all of the tests we write inside of it later when we run the file (e.g. by using "python3 test2.py" from a command line). Further, unittest.TestCase class as an argument gives us access to all of our assertion methods, as well as some useful testing preparation functions like setUp(), which can prepare the program state in advance of executing any tests. Information about this class can (and should!) be read here.

2.3 Anatomy of a Test Case

Your test class will contain several functions that comprise your actual unit tests. These are typically named starting with "test" and take the argument "self". Within a test case, you can write code to set up variables that your assertion functions will test or test against. For example, in this code snippet below, the desired results of the test cases are assigned to variables in order to improve readability.

```
18      # substrings
19      def test_substrings(self):
20          ans1 = {'', 'a', 'b', 'c', 'ab', 'bc', 'abc'}
21          self.assertEqual(substrings('abc'), ans1)
22          ans2 = set(['', 'Boris', 'B', 'i', 'Bor', 'Bo', 'ris', 'o', 'ri', 's',
23                      'r', 'Bori', 'ori', 'oris', 'or', 'is'])
24          self.assertEqual(substrings('Boris'), ans2)
```

The actual meat of these test cases is in the "self.assert..." functions called within the function. These assert functions, outlined in Table 1, provide the framework for determining if your code behaves as intended. They return nothing but when called in the overall execution of the file provide meaningful information about why and how they pass or fail. The usefulness of these functions is not to be underestimated!

Function	Used to check...
assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(x)	bool(x) is True
assertFalse(x)	bool(x) is False
assertIs(a, b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a, b)
assertNotIsInstance(a, b)	not isinstance(a, b)

Table 1: Assert Functions

2.4 Wrapping up the File

```
46  def main():
47      unittest.main()
48
49  if __name__ == '__main__':
50      main()
```

At the end of a test file, you have a main function defined (top level, not contained within the Test... class itself). It is important here to call unittest.main() in main as this actually runs your tests and compiles the output from the individual assertion statements! Finally, add the familiar boilerplate at the end of the file to make sure that it executes when called from the command line.

3 Useful Notes

- Pay attention to the outputs of your tests when they fail! The test file will tell you what assertions were not True or where an error was found.

- Note the distinction between a failure and an error: a failure is your code failing to meet the specification you made in your test file while an error is a bug that causes your code to crash.
- It is usually useful to keep each function to one or two assertions at a max. This makes it easier to recall which behaviors are causing your code to fail/crash.
- Don't forget to change the file from which you are importing when you make a new test file. Usually you're inclined to copy and paste the code from one test file to another, and forgetting this step can lead to some confusing issues.
- Check out the unittest library page for much more information on good testing practice/tools.