CMPSC132 - Programming and Computation II

Department of Computer Science & Engineering
The Pennsylvania State University

1. Unit Testing

In general programming terms, automated testing is the practice of writing code (separate from your actual application code) that invokes the code it tests to help determine if there are any errors. It does not prove that code is correct (which is only possible under very restricted circumstances). It merely reports if the conditions the tester thought of are handled correctly.
Some general rules of testing:

- A testing unit should focus on one tiny bit of functionality and prove it correct.
- Each test unit must be fully independent.
- Learn your tools and learn how to run a single test or a test case. Then, when developing a function inside a module, run this function's tests frequently.
- Always run the full test suite before a coding session, and run it again after. This will give you more confidence that you did not break anything in the rest of the code.
- If you are in the middle of a development session and have to interrupt your work, it is a good idea to write a broken unit test about what you want to develop next. When coming back to work, you will have a pointer to where you were and get back on track faster.
- The first step when you are debugging your code is to write a new test pinpointing the bug. While it is not always possible to do, those bug catching tests are among the most valuable pieces of code in your project.
- Use long and descriptive names for testing functions. The style guide here is slightly different than that of running code, where short names are often preferred. The reason is testing functions are never called explicitly. *square*() or even *power*() is ok in running code, but in testing code you would have names such as *test_square_of_number_2*(), *test_square_negative_number*(). These function names are displayed when a test fails, and should be as descriptive as possible.
- When something goes wrong or has to be changed, and if your code has a good set of tests, you will rely largely on the testing suite to fix the problem or modify a given behavior. Therefore the testing code will be read as much as or even more than the running code.

1.1Python's unittest module

The *unittest* test framework is the batteries-included test module in the Python standard library. The standard workflow is:

1. You define your own class derived from *unittest.TestCase*.
2. Then you fill it with functions that start with 'test_'.
3. You run the tests by placing *unittest.main()* in your file, usually at the bottom.

1.1.1 Basic Example

The *unittest* module provides a rich set of tools for constructing and running tests. This section demonstrates that a small subset of the tools suffice to meet the needs of most users.

Here is a short script to test three string methods:

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

A test case is created by subclassing *unittest.TestCase*. The three individual tests are defined with methods whose names start with the letters test. This naming convention informs the test runner about which methods represent tests.

The crux of each test is a call to *assertEqual*() to check for an expected result; *assertTrue()* or *assertFalse()* to verify a condition; or *assertRaises*() to verify that a specific exception gets raised. These methods are used instead of the assert statement so the test runner can accumulate all test results and produce a report.

The final block shows a simple way to run the tests. *unittest.main*() provides a command-line interface to the test script. When run from the command line, the above script produces an output that looks like this:

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

The *TestCase* class provides several assert methods to check for and report failures. The following table lists the most commonly used methods (see the tables below for more assert methods):

Method	Checks that
`assertEqual(a, b)`	`a == b`
`assertNotEqual(a, b)`	`a != b`

Method	Checks that
`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)`
`assertAlmostEqual(a, b)`	`round(a-b, 7) == 0`
`assertNotAlmostEqual(a, b)`	`round(a-b, 7) != 0`
`assertGreater(a, b)`	`a > b`
`assertGreaterEqual(a, b)`	`a >= b`
`assertLess(a, b)`	`a < b`
`assertLessEqual(a, b)`	`a <= b`
`assertRegex(s, r)`	`r.search(s)`
`assertNotRegex(s, r)`	`not r.search(s)`
`assertCountEqual(a, b)`	`a and b have the same elements in the same number, regardless of their order`

All the assert methods accept a *msg* argument that, if specified, is used as the error message on failure

```
self.assertEqual(s.split(), ['hello', 'world'] , "Failed test for 'Hello world'")
```

References

https://docs.python.org/3/library/unittest.html