

CSCI 553: Networking III

Unix Network Programming

Spring 2007



Unit Testing

Preparation for Testing

Examples

Untar examples

- `$ mkdir testing`
- `$ cd testing`
- `$ tar xvfz /home/csci553/classfiles/testing.tgz`
- Set environment variables for Java & JUnit
 - `$ vi ~/.bashrc`
 - `export PATH=/usr/local/jdk1.6.0/bin:$PATH`
 - `export CLASSPATH=/usr/local/jdk1.6.0/lib/junit-4.4.jar:.`
 - `$. ~/.bashrc`
 - `$ echo $PATH`
 - `$ echo $CLASSPATH`
 - `$ make clean`
 - `$ make testadd`



Introduction

- Unit testing follows a pattern
 - Setup and teardown
 - Lots of small, independent tests
 - Reporting
 - Combine tests into test suites, and test suites into larger suites
- See a pattern, build a framework
 - Write shared code once
 - Encourage people to work a certain way
 - I.e., make it easy for them to do things right



JUnit and Its Children

- JUnit is a testing framework originally written by Kent Beck and Erich Gamma in 1997
 - Made testing easy enough that programmers actually started doing it
 - Now integrated into almost all Java IDEs
- Widely imitated:
 - Workalikes are available C++, Perl, .NET, etc.
 - Once you know one, you can easily learn and use the others
 - Add-ons for measuring test execution times, recording tests, testing web applications, etc.
- This lecture introduces Python's version, called unittest
 - And I'll try and do parallel examples w/ JUnit



The Big Idea

- Define one method for each test
 - Method name must begin with "test"
 - Method must not take any parameters (other than self)
 - Shouldn't return anything
- Group related tests together in classes
 - Which must be derived from `unittest.TestCase`
- Call `unittest.main()`, which:
 - Searches the module (i.e., the file) to find all classes derived from `unittest.TestCase`
 - Runs methods whose names begin with "test" in an arbitrary order
 - Another reason not to make tests dependent on each other
 - Counts and reports the passes, fails, and errors



Checking

- Actually check things inside test methods using methods provided by TestCase
 - Allows the framework to distinguish between test assertions, and normal assert statements
 - Since the code being tested might use the latter
- Checking methods include:
 - `assert_(condition)`: check that something is true (note the underscore)
 - `assertEqual(a, b)`: check that two things are equal
 - `assertNotEqual(a, b)`: the reverse of the above
 - `assertRaises(exception, func, ...args...)`: call `func` with arguments (if provided), and check that it raises the right exception
 - `fail()`: signal an unconditional failure



Example: Checking Addition

```
import unittest

class TestAddition(unittest.TestCase):
    def test_zeroes(self):
        self.assertEqual(0 + 0, 0)
        self.assertEqual(5 + 0, 5)
        self.assertEqual(0 + 13.2, 13.2)
    def test_positive(self):
        self.assertEqual(123 + 456, 579)
        self.assertEqual(1.2e20 + 3.4e20, 3.5e20)
    def test_mixed(self):
        self.assertEqual(-19 + 20, 1)
        self.assertEqual(999 + -1, 998)
        self.assertEqual(-300.1 + -400.2, -700.3)

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

```
.F.
=====
FAIL: test_positive (__main__.TestAddition)
-----
Traceback (most recent call last):
  File "test_addition.py", line 12, in test_positive
    self.assertEqual(1.2e20 + 3.4e20, 3.5e20)
AssertionError: 4.6e+20 != 3.5e+20
-----
Ran 3 tests in 0.000s

FAILED (failures=1)
```



Running Sums

- You want to test a function that calculates a running sum of the values in the list
 - Given $[a, b, c, \dots]$, it produces $[a, a+b, a+b+c, \dots]$
- Test cases:
 - Empty list
 - Single value
 - Long list with mix of positive and negative values
- Hm...is it supposed to:
 - Return a new list?
 - Modify its argument in place and return that?
 - Modify its argument and return None?
- Your tests can only ever be as good as (your understanding of) the spec
 - Assume for now that it's supposed to return a new list



Flawed Implementation

```
def running_sum(seq):
    result = seq[0:1]
    for i in range(2, len(seq)):
        result.append(result[i-1] + seq[i])
    return result

class SumTests(unittest.TestCase):
    def test_empty(self):
        self.assertEqual(running_sum([]), [])
    def test_single(self):
        self.assertEqual(running_sum([3]), [3])
    def test_double(self):
        self.assertEqual(running_sum([2, 9]), [2, 11])
    def test_long(self):
        self.assertEqual(running_sum([-3, 0, 3, -2,
5]), [-3, -3, 0, -2, 3])
```

F.E.

```
=====
ERROR: test_long (__main__.SumTests)
-----
```

Traceback (most recent call last):

File "running_sum_wrong.py", line 22, in test_long

```
self.assertEqual(running_sum([-3, 0, 3, -2, 5]), [-3, -3, 0, -3])
```

File "running_sum_wrong.py", line 7, in running_sum

```
result.append(result[i-1] + seq[i])
```

IndexError: list index out of range

```
=====
FAIL: test_double (__main__.SumTests)
-----
```

Traceback (most recent call last):

File "running_sum_wrong.py", line 19, in test_double

```
self.assertEqual(running_sum([2, 9]), [2, 11])
```

AssertionError: [2] != [2, 11]

```
-----
Ran 4 tests in 0.001s
```

FAILED (failures=1, errors=1)



Check and Re-Check

- Fix the function and rerun the tests

```
def running_sum(seq):  
    result = seq[0:1]  
    for i in range(1, len(seq)):  
        result.append(result[i-1] + seq[i])  
    return result
```

....

Ran 4 tests in 0.000s

OK

- Most first attempts to fix bugs are wrong, or introduce new bugs [McConnell 2004]
 - Continuous testing catches these mistakes while they're still fresh



Is This Cost-Effective

- Should you really go to this much effort to test a simple function?
 - Took less than a minute to write the four tests
 - Uncovered one gap in the requirements, and one error in the first implementation
 - Able to verify the fix almost instantly
 - Sounds pretty good to me...



Another Example

- One More example
- Implement function to reverse a string



Eliminating Redundancy

- Setting up a fixture can often be more work than writing the test
 - The more complex the data structures, the less often you want to have to type them in
- If the test class defines a setUp method, unittest calls it before running each test
 - And if there's a tearDown method, it is run after each test
- Example: test a method that removes atoms from molecules



Eliminating Redundancy

```
class TestThiamine(unittest.TestCase):

    def setUp(self):
        self.fixture = Molecule(C=12, H=20, O=1, N=4,
                                  S=1)
    def test_erase_nothing(self):
        nothing = Molecule()
        self.fixture.erase(nothing)
        self.assertEqual(self.fixture['C'], 12)
        self.assertEqual(self.fixture['H'], 20)
        self.assertEqual(self.fixture['O'], 1)
        self.assertEqual(self.fixture['N'], 4)
        self.assertEqual(self.fixture['S'], 1)
    def test_erase_single(self):
        self.fixture.erase(Molecule(H=1))
        self.assertEqual(self.fixture, Molecule(C=12,
                                                  H=19, O=1, N=4, S=1))
    def test_erase_self(self):
        self.fixture.erase(self.fixture)
        self.assertEqual(self.fixture, Molecule())
```

```
.E.
-----
ERROR: test_erase_self (__main__.TestThiamine)
-----
Traceback (most recent call last):
  File "setup.py", line 49, in test_erase_self
    self.fixture.erase(self.fixture)
  File "setup.py", line 21, in erase
    for k in other.atoms:
RuntimeError: dictionary changed size during iteration
-----
Ran 3 tests in 0.000s

FAILED (errors=1)
```



Testing Exceptions

- Testing that code fails in the right way is just as important as testing that it does the right thing
 - Otherwise, someone will do something wrong some day, and the code won't report it
- In Python, use `TestCase.assertRaises` to check that a specific function raises a specific exception
- In most languages, have to use `try/except` yourself
 - Run the test
 - If execution goes on past it, it didn't raise an exception at all (failure)
 - If the right exception is caught, the test passed
 - If any other exception is caught, the test failed

Manual Exception Testing

Example

- Example: manually test error handling in a function that finds all values in a double-ended range
 - Raises `ValueError` if the range is empty, or if the set of values is empty

```
class TestInRange(unittest.TestCase):
    def test_no_values(self):
        try:
            in_range([], 0.0, 1.0)
        except ValueError:
            pass
        else:
            self.fail()
    def test_bad_range(self):
        try:
            in_range([0.0], 4.0, -2.0)
        except ValueError:
            pass
        else:
            self.fail()
```



Testing I/O

- Input and output often seem hard to test
 - Store a bunch of input files in a subdirectory?
 - Create temporary files when tests are run?
- The best answer is to use I/O using strings
 - Python's StringIO and cStringIO modules can read and write strings instead of files
 - Similar packages exist for C++, Java, and other languages
- This only works if the function being tested takes streams as arguments, rather than filenames
 - If the function opens and closes the file, no way for you to substitute a fake file
 - You have to design code to make it testable

I/O Testing Example

```
class TestDiff(unittest.TestCase):

    def wrap_and_run(self, left, right, expected):
        left = StringIO(left)
        right = StringIO(right)
        actual = StringIO()
        diff(left, right, actual)
        self.assertEqual(actual.getvalue(), expected)

    def test_empty(self):
        self.wrap_and_run('', '', '')

    def test_lengthy_match(self):
        str = '''\
a
b
c
'''
        self.wrap_and_run(str, str, '')

    def test_single_line_mismatch(self):
        self.wrap_and_run('a\n', 'b\n', '1\n')

    def test_middle_mismatch(self):
        self.wrap_and_run('a\nb\nc\n', 'a\nx\nc\n', '2\n')
```

- Example: find lines where two files differ
 - Input: two streams (which might be open files or StringIO wrappers around strings)
 - Output: another stream (i.e., a file, or a StringIO)
- As a side effect, we've made the function itself more useful
 - People can now use it to compare strings to strings, or strings to files



Stubs and Mock Objects

- A stub is a placeholder for a function or method you haven't written yet
 - Always returns the same value (or a random one)
 - Created so that you don't have to wait until your whole program is written before running and testing it
 - Eventually replaced with real code
- Mock objects are more sophisticated
 - Has the same interface as the object whose place it takes
 - But return values of methods are hard-coded
 - E.g., use a dictionary of possible argument values to look up the correct response, instead of consulting a database
 - Used to isolate components during testing
 - Use a real instance of the object under suspicion, and mock replacements for everything else
 - Not thrown away once the program is working



Test Performance

- Making tests run fast is another reason to use stubs, mock objects, and other tricks
 - Reinitializing a database on disk can take 1-2 seconds
 - So 500 tests take 10 minutes to run
 - Makes it impractical for developers can't re-run the tests after every small code change
- “Test performance” can also mean “test how fast the target code is”
 - Record how long it takes to run the test suite
 - Sudden increases or decreases may signal bugs
 - Even if they don't, you probably want to know that your code is four times slower than it used to be before you ship it



Choosing Test Cases

- Human beings are creatures of habit
 - Tend to make the same kinds of errors over and over again
 - So test for those first
 - Once you start testing for habitual errors, you become more conscious of them, and make them less often
- A catalog of errors
 - Numbers: zero, largest, smallest magnitude, most negative
 - Structures: empty, exactly one element, maximum number of elements
 - Duplicate elements (e.g., the letter "J" appears three times in a string)
 - Aliased elements (e.g., a list contains two references to another list)
 - Circular structures (e.g., a list that contains a reference to itself)
 - Searching: no match found, one match found, multiple matches found, everything matches
 - Code like `x = find_all(structure)[0]` is almost always wrong
 - Should also check aliased matches (same thing found multiple times)



Example: Rectangle Overlap

- Want to test a function that calculates the overlap between two rectangles

Solution: Rectangle Overlap

- Assume for the moment that Rect is correct
 - I.e., that it has been tested elsewhere
- Each fixture will be a pair of rectangles
 - The test will be to pass them to overlap, and see if the output is correct
- In this example, “boundary case” and “corner case” can be taken literally

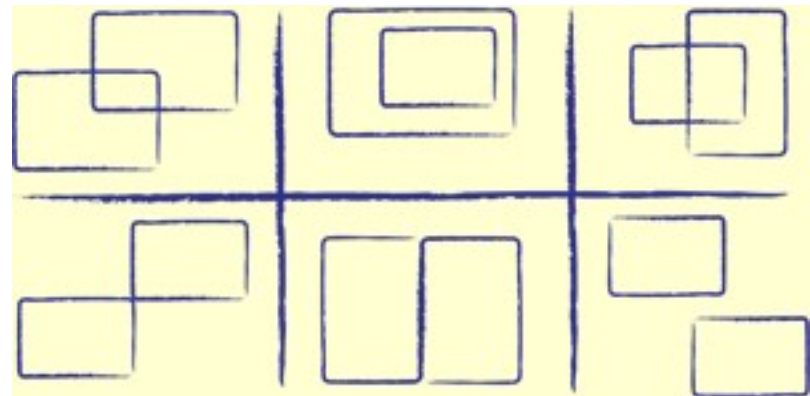


Figure 1: Rectangle Overlap Test Case



What Tests to Write First

- Tests you expect to succeed
 - Boundary cases (e.g., sort the empty list, or a list of one value)
 - Simplest interesting case (e.g., sort a list of two values)
 - General case (e.g., sort a list of nine values)
 - If duplicate values are allowed, make sure you test with them
- Tests you expect to fail
 - Invalid input (e.g., passed a dictionary instead of a list)
 - Remember, error handling is part of the interface too
- Sanity tests
 - Make sure data structures remain consistent
 - If there is redundant information, check it against itself



Summary

- A good framework does more than just cut down on typing
 - Guides you toward solutions that other developers have already discovered
- The better you are at testing (and using testing frameworks), the more productive you will be