Testing

Clancy Rowley and Robert Lupton

2014-07-27

Testing vs. debugging

Testing

- Orderly, systematic, exhaustive checks on whether a program is working properly

Testing vs. debugging

Testing

- Orderly, systematic, exhaustive checks on whether a program is working properly
  - test as you write the code
  - test systematically
  - automate testing

Testing vs. debugging

Testing

- Orderly, systematic, exhaustive checks on whether a program is working properly
  - test as you write the code
  - test systematically
  - automate testing
- Testing process begins when program is designed
  - design an interface with testing in mind
  - continues as program evolves
  - never ends

Testing vs. debugging

Testing

- Orderly, systematic, exhaustive checks on whether a program is working properly
  - test as you write the code
  - test systematically
  - automate testing
- Testing process begins when program is designed
  - design an interface with testing in mind
  - continues as program evolves
  - never ends

Debugging

- Efficiently figuring out what’s wrong

Testing vs. debugging

Testing

- Orderly, systematic, exhaustive checks on whether a program is working properly
  - test as you write the code
  - test systematically
  - automate testing
- Testing process begins when program is designed
  - design an interface with testing in mind
  - continues as program evolves
  - never ends

Debugging

- Efficiently figuring out what's wrong
- never ends either

Test first or last?

Test last

- Write a code, then write tests to find the bugs

Test first or last?

Test last

- Write a code, then write tests to find the bugs

Test first (TDD: Test Driven Development)

- Write tests *before* writing the code being tested
- Advantages
  - Ensures tests are working (all should fail initially)
  - See progress as you implement more features
  - Can detect problems with the interface before spending time on an implementation (and possibly coding the wrong thing)

Test as you write the code

Many errors can be eliminated before they happen, by careful coding

- Check boundary conditions
- Look for off-by-one errors
- State and check pre-conditions
- Add assertions for post-conditions and invariants
- Defensive programming
- Check error returns
- Turn on all compiler checks

Systematic testing

Test incrementally

- Test each new feature as it's implemented
- Test each new code as it's written

Systematic testing

Test incrementally

- Test each new feature as it's implemented
- Test each new code as it's written

Test simple parts first

- Basic features before fancy ones

Systematic testing

Test incrementally

- Test each new feature as it's implemented
- Test each new code as it's written

Test simple parts first

- Basic features before fancy ones

Know what output to expect

- Easiest if test inputs and outputs are in files and can be run automatically

Systematic testing

Test incrementally

- Test each new feature as it's implemented
- Test each new code as it's written

Test simple parts first

- Basic features before fancy ones

Know what output to expect

- Easiest if test inputs and outputs are in files and can be run automatically

Verify conservation properties

- Preserves number of items, sizes of data structures, order relationships, etc

Systematic testing

Test incrementally

- Test each new feature as it's implemented
- Test each new code as it's written

Test simple parts first

- Basic features before fancy ones

Know what output to expect

- Easiest if test inputs and outputs are in files and can be run automatically

Verify conservation properties

- Preserves number of items, sizes of data structures, order relationships, etc

Compare independent implementations

- Brute force, inefficient version with more complex, fast version

Systematic testing

Test incrementally

- Test each new feature as it's implemented
- Test each new code as it's written

Test simple parts first

- Basic features before fancy ones

Know what output to expect

- Easiest if test inputs and outputs are in files and can be run automatically

Verify conservation properties

- Preserves number of items, sizes of data structures, order relationships, etc

Compare independent implementations

- Brute force, inefficient version with more complex, fast version

Measure test coverage

- Use tools such as `tcov` to generate coverage information
- Add test cases to go through each branch of code
- It's hard to achieve high coverage

Mechanization: let the computer do the work

Write code by programs

- specialized languages: regular expressions,. . .

Use executable specifications

- avoid writing the same information multiple times
- capture the varying part in specification files

Do clerical jobs by programs

- build, test, system administration,. . .

Test programs by programs

- hand testing is too slow and prone to error

Main point

When the job is mechanical, it’s time to mechanize.

Example: testing binary search

```
while True:
    line = raw_input("Enter size, item: ")
    try:
        n, item = [int(i) for i in line.split()]
    except:
        break
    x = [10 * a for a in range(n)]
    print " %d" % binary_search(x, item)
```

Example: testing binary search

```
while True:
    line = raw_input("Enter size, item: ")
    try:
        n, item = [int(i) for i in line.split()]
    except:
        break
    x = [10 * a for a in range(n)]
    print " %d" % binary_search(x, item)
```

Usage:

```
Enter size, item: 5 0
 0
Enter size, item: 5 10
 1
Enter size, item: 5 40
 4
Enter size, item: 5 5
 -1
Enter size, item: 5 -5
 -1
Enter size, item: 5 50
 -1
Enter size, item: 5 45
 -1
```

Test automation

Test scaffolds

- shell scripts, programs, makefiles, etc, that run tests and compare results automatically

Automated regression testing

- Does the new version get the same answer as the old version?
- And in about the same amount of time?
- If it doesn't, explain why

Self-contained tests

- Compare to known output or independently created outputs

Stress tests

- large inputs
- random inputs
- perverse inputs

Example: testing AWK

Brian Kernighan maintains the original version of AWK, and a comprehensive test suite

- Around 1000 tests, run with a single command
- Regression tests
  - Does it work the same as the previous version does?
- Known output tests
  - Does it produce the same output as an independent mechanism?
- Bug fix tests
  - A test for each bug and new feature
- Stress tests
  - Does it handle perverse, huge, illegal, random cases?
- Coverage tests
  - Are all statements executed by some test?
- Performance tests
  - Does it run at the same speed or better?

Using AWK for testing regexp code

- Regular expression tests are described in a very small specialized language

```
^a.$ ~   ax
         aa
    !~   xa
         aaa
         axy
```

- Each test is converted into a command that exercises the new version:

```
$ echo 'ax' | awk '!/^a.$/ {print "bad"}'
```

- Illustrates
  - little languages
  - programs that write programs
  - mechanization

Python version

Automation of binary search test, version 1

```
tests = """\
0 0 -1
1 0 0
1 10 -1
1 -5 -1
5 0 0
5 10 1
5 40 4
5 5 -1
5 -5 -1
5 50 -1
5 45 -1"""
for line in tests.split('\n'):
    (size, item, expected) = [int(i) for i in line.split()]
    x = [10 * i for i in range(size)]
    assert(binary_search(x, item) == expected)
```

Automation of binary search test, version 1

```
tests = """\
0 0 -1
1 0 0
1 10 -1
1 -5 -1
5 0 0
5 10 1
5 40 4
5 5 -1
5 -5 -1
5 50 -1
5 45 -1"""
for line in tests.split('\n'):
    (size, item, expected) = [int(i) for i in line.split()]
    x = [10 * i for i in range(size)]
    assert(binary_search(x, item) == expected)
```

- Nice, compact language; easy to specify new tests
- Still somewhat error prone: need to list each test by hand

Automation of binary search test, version 2

Generate an array of a given size, and try searching for *all* elements:

```
for size in range(100):
    # test array of the form [0, 10, 20, 30, ...]
    x = [10 * i for i in range(size)]
    for i in range(size):
        assert(binary_search(x, x[i]) == i)
        assert(binary_search(x, x[i] - 5) == -1)
    assert(binary_search(x, 10*size) == -1)
    assert(binary_search(x, 10*size - 5) == -1)
```

Automation of binary search test, version 2

Generate an array of a given size, and try searching for *all* elements:

```
for size in range(100):
    # test array of the form [0, 10, 20, 30, ...]
    x = [10 * i for i in range(size)]
    for i in range(size):
        assert(binary_search(x, x[i]) == i)
        assert(binary_search(x, x[i] - 5) == -1)
    assert(binary_search(x, 10*size) == -1)
    assert(binary_search(x, 10*size - 5) == -1)
```

Test arrays with equal elements:

```
    # test an array with equal elements: [10, 10, 10, ...]
    x = size * [10]
    if size == 0:
        assert(binary_search(x, 10) == -1)
    else:
        assert(binary_search(x, 10) in range(size))
    assert(binary_search(x, 5) == -1)
    assert(binary_search(x, 15) == -1)
```

Automation of binary search test, version 2

Generate an array of a given size, and try searching for *all* elements:

```
for size in range(100):
    # test array of the form [0, 10, 20, 30, ...]
    x = [10 * i for i in range(size)]
    for i in range(size):
        assert(binary_search(x, x[i]) == i)
        assert(binary_search(x, x[i] - 5) == -1)
    assert(binary_search(x, 10*size) == -1)
    assert(binary_search(x, 10*size - 5) == -1)
```

Test arrays with equal elements:

```
    # test an array with equal elements: [10, 10, 10, ...]
    x = size * [10]
    if size == 0:
        assert(binary_search(x, 10) == -1)
    else:
        assert(binary_search(x, 10) in range(size))
    assert(binary_search(x, 5) == -1)
    assert(binary_search(x, 15) == -1)
```

- Compact, clear code.

Automation of binary search test, version 2

Generate an array of a given size, and try searching for *all* elements:

```
for size in range(100):
    # test array of the form [0, 10, 20, 30, ...]
    x = [10 * i for i in range(size)]
    for i in range(size):
        assert(binary_search(x, x[i]) == i)
        assert(binary_search(x, x[i] - 5) == -1)
    assert(binary_search(x, 10*size) == -1)
    assert(binary_search(x, 10*size - 5) == -1)
```

Test arrays with equal elements:

```
    # test an array with equal elements: [10, 10, 10, ...]
    x = size * [10]
    if size == 0:
        assert(binary_search(x, 10) == -1)
    else:
        assert(binary_search(x, 10) in range(size))
    assert(binary_search(x, 5) == -1)
    assert(binary_search(x, 15) == -1)
```

- Compact, clear code.
- Drawback: `assert` stops execution after first test fails. Potentially need to run the test code *n* times to find *n* bugs.
- Would be nice to keep a count of number of tests passed/failed.

Automation of binary search test, version 3

Keep a count of number of tests passed and failed.

Function to check a single element, and print an error message if `binary_search` does not return the expected value:

```
num_tests_passed = 0
num_tests_failed = 0

def check(condition, msg):
    """Record test passed or failed if status True/False,
    and print msg if failed"""
    global num_tests_passed
    global num_tests_failed
    if condition:
        num_tests_passed += 1
    else:
        num_tests_failed += 1
        print "Test failed: %s" % msg

def check_item(x, item, expected):
    """Check that binary_search(x, item) == expected"""
    ind = binary_search(x, item);
    msg = "binary_search(x, %d) = %d, expected %d\n  x = %s" \
          % (item, ind, expected, str(x))
    check(ind == expected, msg)
```

Automation of binary search test, version 3

Driver program:

```
for size in range(100):
    # test array of the form [0, 10, 20, 30, ...]
    x = [10 * i for i in range(size)]
    for i in range(size):
        check_item(x, x[i], i)
        check_item(x, x[i] - 5, -1)
    check_item(x, 10*size, -1)
    check_item(x, 10*size - 5, -1)

    # test an array with equal elements: [10, 10, 10, ...]
    x = size * [10]
    if size == 0:
        check_item(x, 10, -1)
    else:
        ind = binary_search(x, 10)
        msg = "binary_search(x, 10) = %d, expected [0..%d)\n  x = %s" \
            % (ind, size, str(x))
        check(ind in range(size), msg)
    check_item(x, 5, -1)
    check_item(x, 15, -1)
print "%d passed, %d failed" % (num_tests_passed, num_tests_failed)
```

Automation of binary search test, version 3

Driver program:

```
for size in range(100):
    # test array of the form [0, 10, 20, 30, ...]
    x = [10 * i for i in range(size)]
    for i in range(size):
        check_item(x, x[i], i)
        check_item(x, x[i] - 5, -1)
    check_item(x, 10*size, -1)
    check_item(x, 10*size - 5, -1)

    # test an array with equal elements: [10, 10, 10, ...]
    x = size * [10]
    if size == 0:
        check_item(x, 10, -1)
    else:
        ind = binary_search(x, 10)
        msg = "binary_search(x, 10) = %d, expected [0..%d)\n  x = %s" \
            % (ind, size, str(x))
        check(ind in range(size), msg)
    check_item(x, 5, -1)
    check_item(x, 15, -1)
print "%d passed, %d failed" % (num_tests_passed, num_tests_failed)
```

- Would be better to replace global variable by a variable inside a class

Automation of binary search test, version 3

Sample output (buggy version):

```
Test failed: binary_search(x, 0) = -1, expected 0
  x = [0]
Test failed: binary_search(x, 10) = -1, expected [0..1)
  x = [10]
Test failed: binary_search(x, 10) = -1, expected 1
  x = [0, 10]
Test failed: binary_search(x, 0) = -1, expected 0
  x = [0, 10, 20]
Test failed: binary_search(x, 20) = -1, expected 2
  x = [0, 10, 20]
27 passed, 5 failed
```

Sample output (working version):

```
32 passed, 0 failed
```

Automation of binary search test, version 4 I

```
class Tester(object):
    _max_size = 100

    def __init__(self):
        self._num_passed = 0
        self._num_failed = 0

    def _check(self, condition, msg):
        """Record test passed or failed, and print msg if failed"""
        if condition:
            self._num_passed += 1
        else:
            self._num_failed += 1
            print "Test failed: %s" % msg

    def _check_item(self, x, item, expected):
        """Check that binary_search(x, item) == expected"""
        ind = binary_search(x, item);
        msg = "binary_search(x, %d) = %d, expected %d\n  x = %s" \
            % (item, ind, expected, str(x))
        self._check(ind == expected, msg)

    def test_array(self):
        """Check all possible values in an array [0, 10, 20,...]"""
        for size in range(self._max_size):
            x = [10 * i for i in range(size)]
            for i in range(size):
                self._check_item(x, x[i], i)
                self._check_item(x, x[i] - 5, -1)
```

Automation of binary search test, version 4 II

```
            self._check_item(x, 10*size, -1)
            self._check_item(x, 10*size - 5, -1)

    def test_equal_elems(self):
        """Check values in an array [10, 10, 10,...]"""
        for size in range(self._max_size):
            x = size * [10]
            if size == 0:
                self._check_item(x, 10, -1)
            else:
                ind = binary_search(x, 10)
                msg = "binary_search(x, 10) = %d, expected [0,%d)\n  x = %s" \
                    % (ind, size, str(x))
                self._check(ind in range(size), msg)
            self._check_item(x, 5, -1)
            self._check_item(x, 15, -1)

    def print_status(self):
        print "%d passed, %d failed" % (self._num_passed, self._num_failed)

    def run_tests(self):
        self.test_array()
        self.test_equal_elems()
        self.print_status()
```

Driver program:

```
Tester().run_tests()
```

JUnit family of test frameworks

- Java has a built-in facility for automating tests: JUnit
- It has been copied in several other languages:
  - Google C++ testing framework: `http://code.google.com/p/googletest/`
  - python `unittest` (part of standard library)
- Features
  - Choose different output formats for test results
  - Can write set-up and tear-down functions that are called before and after each test
  - Object-oriented (use inheritance to avoid duplicate test code)
  - Fairly standardized

Automation of binary search test, version 5 I

```
import unittest

class TestBinarySearch(unittest.TestCase):
    def setUp(self):
        self.sizes = range(100)

    def check_item(self, x, item, expected):
        result = binary_search(x, item)
        msg = "binary_search(x, %d) = %d, expected %d\nx = %s" % \
              (item, result, expected, str(x))
        self.assertEqual(result, expected, msg)

    def test_all(self):
        """Search for all items in an array"""
        for size in self.sizes:
            x = [10 * i for i in range(size)]
            for i in range(size):
                self.check_item(x, x[i], i)
                self.check_item(x, x[i] - 5, -1)
            self.check_item(x, 10*size, -1)
            self.check_item(x, 10*size - 5, -1)

    def test_equal_elems(self):
        """Search an array whose elements are all equal"""
        for size in self.sizes:
            x = size * [10]
            if size == 0:
                self.check_item(x, 10, -1)
            else:
```

Automation of binary search test, version 5 II

```
            result = binary_search(x, 10)
            msg = "binary_search(x, %d) = %d, expected [0,%d)\nx = %s" % \
                  (10, result, size, x)
            self.assert_(result in range(0,size), msg)
        self.check_item(x, 5, -1)
        self.check_item(x, 15, -1)
```

Driver program:

```
unittest.main()
```

Output (working version):

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.029s

OK
```

Automation of binary search test, version 5

Output (buggy version):

```
FF
======================================================================
FAIL: test_all (__main__.TestBinarySearch)
Search for all items in an array
----------------------------------------------------------------------
Traceback (most recent call last):
  File "src/binsearch.py", line 238, in test_all
    self.check_item(x, x[i], i)
  File "src/binsearch.py", line 231, in check_item
    self.assertEqual(result, expected, msg)
AssertionError: binary_search(x, 0) = -1, expected 0
x = [0]

======================================================================
FAIL: test_equal_elems (__main__.TestBinarySearch)
Search an array whose elements are all equal
----------------------------------------------------------------------
Traceback (most recent call last):
  File "src/binsearch.py", line 253, in test_equal_elems
    self.assert_(result in range(0,size), msg)
AssertionError: binary_search(x, 10) = -1, expected [0,1)
x = [10]

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=2)
```

Skipping Tests

It’s often convenient to skip some tests; the best way to do this is using one of `unittest`’s decorators:

```
@unittest.skip("demonstrating skipping")
def test_nothing(self):
    self.fail("shouldn't happen")
```

There are also `@unittest.skipIf` and `@unittest.skipUnless`; they can also be used to skip entire classes of tests.

Skipping Tests

It's often convenient to skip some tests; the best way to do this is using one of `unittest`'s decorators:

```
@unittest.skip("demonstrating skipping")
def test_nothing(self):
    self.fail("shouldn't happen")
```

There are also `@unittest.skipIf` and `@unittest.skipUnless`; they can also be used to skip entire classes of tests.

Another useful decorator is `@unittest.expectedFailure` — especially useful as it's all too easy to keep skipping tests even after you've made them pass.

Python version of AWK regexp test

- Recall compact tests for regular expressions in AWK

Previous slide

Python version of AWK regexp test

- Recall compact tests for regular expressions in AWK

Previous slide

- Can do similar things with `unittest` in Python:

```
#!/usr/bin/env python
import re, unittest

class RegexpTestCase(unittest.TestCase):
   """A test fixture for regexp products"""

   def testRe(self):
       tests = {"^a.$" : (["ax", "aa"], ["xa", "aaa", "axy"]),
                }

       for p in tests.keys():
           good, bad = tests[p]
           for s in good:
               self.assertTrue(re.search(p, s))
           for s in bad:
               self.assertFalse(re.search(p, s))

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

Issues with testing scientific codes

- Floating point doesn't give exact answers
  - Use `assertAlmostEqual()` or equivalent (`EXPECT_DOUBLE_EQ()` for Google C++ test)
- Always compare against a known answer
  - Not always easy for scientific codes: even if we know an exact solution for a particular problem, our numerical methods are often only approximations. What should our error tolerance be?
  - If you know a numerical method is exact for a certain type of input (for instance, an interpolation algorithm on a collinear set of points), use that as a test problem.
  - Comparing against a previous, trusted version can be useful here

Unit tests *v.* Integration tests *v.* Regression tests

There are a number of types of testing that you need to worry about:

Unit tests *v.* Integration tests *v.* Regression tests

There are a number of types of testing that you need to worry about:

- Unit testing. Do your pieces of code (classes, functions) work correctly?

Unit tests *v.* Integration tests *v.* Regression tests

There are a number of types of testing that you need to worry about:

- Unit testing. Do your pieces of code (classes, functions) work correctly?
- Integration testing. Do the units work properly together?

Unit tests *v.* Integration tests *v.* Regression tests

There are a number of types of testing that you need to worry about:

- Unit testing. Do your pieces of code (classes, functions) work correctly?
- Integration testing. Do the units work properly together?
- Regression testing: Do things still work?

Unit tests *v.* Integration tests *v.* Regression tests

There are a number of types of testing that you need to worry about:

- Unit testing. Do your pieces of code (classes, functions) work correctly?
- Integration testing. Do the units work properly together?
- Regression testing: Do things still work?
- Acceptance tests: Will you be paid/graduate/get tenure?

Unit tests *v.* Integration tests *v.* Regression tests

There are a number of types of testing that you need to worry about:

- Unit testing. Do your pieces of code (classes, functions) work correctly?
- Integration testing. Do the units work properly together?
- Regression testing: Do things still work?
- Acceptance tests: Will you be paid/graduate/get tenure?

Because unit tests are supposed to test code locally, you may want to *mock* other parts of the system (*e.g.* if you're testing an object finder that works in smoothed images, you don't want to test the I/O and convolution parts of the code).

Unit tests *v.* Integration tests *v.* Regression tests

There are a number of types of testing that you need to worry about:

- Unit testing. Do your pieces of code (classes, functions) work correctly?
- Integration testing. Do the units work properly together?
- Regression testing: Do things still work?
- Acceptance tests: Will you be paid/graduate/get tenure?

Because unit tests are supposed to test code locally, you may want to *mock* other parts of the system (*e.g.* if you're testing an object finder that works in smoothed images, you don't want to test the I/O and convolution parts of the code). In python, `unittest.mock` makes this (relatively) easy;

Unit tests *v.* Integration tests *v.* Regression tests

There are a number of types of testing that you need to worry about:

- Unit testing. Do your pieces of code (classes, functions) work correctly?
- Integration testing. Do the units work properly together?
- Regression testing: Do things still work?
- Acceptance tests: Will you be paid/graduate/get tenure?

Because unit tests are supposed to test code locally, you may want to *mock* other parts of the system (*e.g.* if you're testing an object finder that works in smoothed images, you don't want to test the I/O and convolution parts of the code). In python, `unittest.mock` makes this (relatively) easy; in python3, available for python 2.7.

Continuous Integration (*CI*)

Rather than run your tests by hand, it's convenient to use some automated framework.

Continuous Integration (*CI*)

Rather than run your tests by hand, it's convenient to use some automated framework. Unit tests are easily run as part of your regular build (*e.g.* via a `make test` target in a Makefile), but in larger systems you probably want to run larger scale tests automatically too.

Continuous Integration (*CI*)

Rather than run your tests by hand, it's convenient to use some automated framework. Unit tests are easily run as part of your regular build (*e.g.* via a `make test` target in a Makefile), but in larger systems you probably want to run larger scale tests automatically too.

Popular frameworks are `buildbot`, `jenkins`, and `travis-CI`.

Continuous Integration (*CI*)

Rather than run your tests by hand, it's convenient to use some automated framework. Unit tests are easily run as part of your regular build (*e.g.* via a `make test` target in a Makefile), but in larger systems you probably want to run larger scale tests automatically too.

Popular frameworks are `buildbot`, `jenkins`, and `travis-CI`.

LSST uses `jenkins` to periodically check the state of `master` (and some of the repos use `travis` as a pre-commit check)

Summary

- Test your code
- Automate, automate, automate
- Write programs that write programs: leverage
- Design clean interfaces that make your code easy to test
- Consider writing tests *before* you write your code
- Whenever you find a bug, write a test that exposes it
- Whenever you add a feature, write a test that exercises it
- Look at available frameworks for organizing tests
- Think about using a *CI* framework too