Software Testing

E6891 Lecture 5
2014-02-26

Today's plan

- Overview of software testing
  - adapted from [Software Carpentry]
- Best practices for numerical computation
- Examples

Software testing

- Automatic **failure** detection
- **NOT** “correctness detection”
- Bugs are inevitable, but we’d like to find them quickly
- Do this right, and the tests will dictate program behavior

Why do we need tests?

Correctness

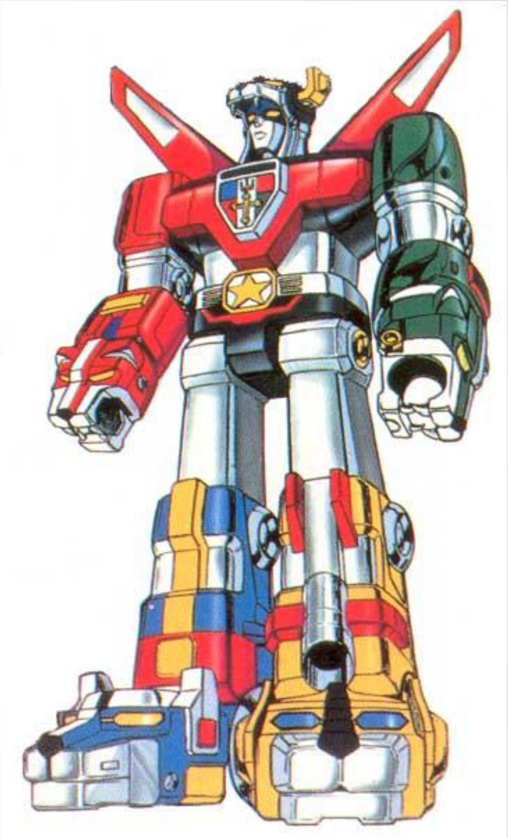
- *Does implementation match specification?*
  - Implementation = code
  - Specification = equation, algorithm, paper, etc.

- Important for both ends of research
  - Accurately reporting your own method
  - Ensuring accurate replication of a reported method

Why do we need tests?

Software design

- Thinking about failure modes improves software design
  - Isolate critical functions
  - Specify behavior
  - Explicit error handling
- Test each component
- End result
  - simplified high-level functions



Why do we need tests?
Debugging

- *Something's wrong in my code!*
- *But submodules X and Y pass tests...*
- *so the bug must be in Z!*
    - well, probably...

Why do we need tests?
Optimization, refactoring

- *My experiments are taking too long!*
- *Maybe I can optimize my algorithm…*
- Is the faster version equivalent?

Unit testing

- Software is built from small components
  - input parser
  - feature extractor
  - number crunching
  - ...
- Don’t try to test the whole thing at once
- Test each component (**unit**) independently

Unit testing

- **Unit**
  - function being tested
- **Fixture**
  - the test input
  - many for each unit
- **Action**
  - How to combine the **unit** + **fixture**
  - that is, **test code**
- **Expected result**
  - What should the **action** produce?
  - **return value?**
  - Maybe an **exception**
- **Actual result**
  - Did it match expectation?

Example: computing norms

- **Unit**
  - ```
    def norm(x, p):
      n = 0
      for xi in x:
        n += xi**p
      n = n**(1.0/p)
      return p
    ```
- **Fixtures + results**
  - `( ([ 1, 0, 0], 1), 1.0)`
  - `( ([ 1, 0, 0], 2), 1.0)`
  - `( ([-1, 0, 0], 1), 1.0)`
  - `( ([-1, 0, 0], 2), 1.0)`
  - `...`
- **Action**
  - ```
    n = norm(fix[0],fix[1])
    assert n == result
    ```

Example: computing norms

- **Unit**
  - ```
    def norm(x, p):
      n = 0
      for xi in x:
        n += xi**p
      n = n**(1.0/p)
      return p
    ```
- **Fixtures + results**
  - `( ([ 1, 0, 0], 1), 1.0)`
  - `( ([ 1, 0, 0], 2), 1.0)`
  - `( ([-1, 0, 0], 1), 1.0)`
  - `( ([-1, 0, 0], 2), 1.0)`
  - ...
- **Action**
  - ```
    n = norm(fix[0],fix[1])
    assert n == result
    ```
- Report
  - Pass
  - Pass
  - Fail
  - Pass
  - ...

Example: computing norms

- Unit
  - ```
    def norm(x, p):
      n = 0
      for xi in x:
        n += xi**p
      n = n**(1.0/p)
      return p
    ```
- Fixtures + results
  - `( ([ 1, 0, 0], 1), 1.0)`
  - `( ([ 1, 0, 0], 2), 1.0)`
  - `( ([-1, 0, 0], 1), 1.0)`
  - `( ([-1, 0, 0], 2), 1.0)`
  - `...`
- Action
  - ```
    n = norm(fix[0],fix[1])
    assert n == result
    ```
- Report
  - Pass
  - Pass
  - Fail
  - Pass
  - ...

Example: computing norms

- **Unit**
  - ```
    def norm(x, p):
        n = 0
        for xi in x:
            n += abs(xi)**p
        n = n**(1.0/p)
        return p
    ```
- **Fixtures + results**
  - `( ([ 1, 0, 0], 1), 1.0)`
  - `( ([ 1, 0, 0], 2), 1.0)`
  - `( ([-1, 0, 0], 1), 1.0)`
  - `( ([-1, 0, 0], 2), 1.0)`
  - `...`
- **Action**
  - ```
    n = norm(fix[0],fix[1])
    assert n == result
    ```
- **Report**
  - Pass
  - Pass
  - Pass
  - Pass
  - ...

Designing test cases

- Exhaustive testing is generally impossible

- But don’t just use a single case either

- Seek out corner cases and assumptions
  - anywhere there’s a condition (if-then-else)
  - calls to other functions

Designing test cases: exercise

```
def norm(x, p):
  n = 0
  for xi in x:
    n += abs(xi)**p
  n = n**(1.0/p)
  return p
```

- What are the assumptions in this code?
- What are good test cases?

Designing test cases: exercise

```
def norm(x, p):
  n = 0
  for xi in x:
    n += abs(xi)**p
  n = n**(1.0/p)
  return p
```

- `p > 0`
- `p finite`
- `len(x) > 0`
- `xi +, -, 0?`
- `xi finite`
- `Others?`

- What are the assumptions in this code?
- What are good test cases?

Success vs failure?

- Tests can only identify incorrect behavior
    - Tests are never 100% complete
- **Failure** is correct behavior if the input is bad
    - Silent failure is a debugging nightmare
    - Use **exceptions**!
    - Even MATLAB has exceptions now...
- If you only test **success** cases, failure may not be identified in practice

Don’t go overboard...

- Not all failures need to be handled
  - what if p is a string?
  - what if x is a matrix?
- Use test cases to guide development
- Testing makes documentation easier

```
def norm(x, p):
  ```Requires:
     type(x) = ndarray
     type(p) = float
     p > 0
  ```
  if p <= 0:
    raise ValueError()
  n = 0
  for xi in x:
    n += abs(xi)**p
  n = n**(1.0/p)
  return p
```

Testing numerical methods

- Some numerical routines are complicated
  - `integral(f, a, b)`
- Tests should depend on the **interface**
  - Not the implementation!
- Try to design test cases with known answers
  - `([f(x) = 1.0, a=0, b=2],      2.0)`
  - `([f(x) = abs(x), a=-1, b=1], 1.0)`

Testing numerical methods

- In floating point, things are rarely identical
- **BAD**: too strict, relies on machine precision
    - `assert f(x) == result`
- **BETTER**: allows small absolute differences
    - `assert abs(f(x) - result) < 1e-10`
- **BEST**: allows small relative differences
    - `assert np.allclose(f(x), result)`

What if solutions are not unique?

- Examples:
  - `sqrt(x)`, positive or negative?
  - eigenvectors
  - k-means, mixture models, etc.
- Don’t test quantitatively
  - `assert np.allclose(sqrt_x, exp_sqrt_x)`
- Test qualitatively
  - `assert np.allclose(sqrt_x**2, x)`

What is *correct* anyway?

- Often, behavior is not clearly specified
  - e.g.: automatic beat tracking
  - no *right* answer for a given input

- Maybe we’re just matching a previous implementation
  - while refactoring or optimizing code
  - or porting/re-implementing in a new language

- Generate fixture/result pairs by running the old version

Testing frameworks

- Writing test code is no fun
- Fortunately, most languages have test suites
  - (yes, even MATLAB)
- We'll talk about python's `nosetest` module

nosetest

- Implement actions as functions `test_*`
- Automatic report generation
- Advanced features
    - exception handling
    - fixture setup/teardown
    - function/class/module/package support
    - test generators: iterate over fixtures

Using nosetest

```
bmcfee@crushinator: ~/git/librosa/tests
[~/git/librosa/tests:develop]➔ nosetests
.........................................................................
.........................................................................
......................................................................
----------------------------------------------------------------------
Ran 310 tests in 6.848s

OK
[~/git/librosa/tests:develop]➔
```

An example: librosa

Wrap up

- Automated testing will make your life easier (in the long run)
- It’s not difficult
- Your code will be better
- No (fewer?) late-night panic attacks