Comp-304 : Testing Strategies
Lecture 3

Alexandre Denault
Original notes by Hans Vangheluwe
Computer Science
McGill University
Fall 2006

Python

Do you know Python yet?

Questions

- How is scope defined?
- What is an object?
- What keyword is used to defined classes?
- In a class, what is the first argument of every function?
- How do you declare a variable?
- What keyword is used to define a constructor?
- What is the name of the main function?

Help!

- If you didn't know the answers to the previous questions, check out:

  Dive Into Python

  http://www.diveintopython.org/

  Python Library Reference

  http://www.python.org/doc/current/lib/lib.html

Bug

- A software bug is an error, flaw, mistake, failure, or fault in a computer program that prevents it from working as intended, or produces an incorrect result.
- Bugs can exist at different levels
  - Design
  - Source Code
- Bugs have severities
  - Some bugs simply crash an application.
  - Some bugs cause loss of data.
  - Some bugs cause loss of money.
  - The worst bugs cause loss of life.

Origin or Bugs

In 1946, when Hopper was released from active duty, she joined the Harvard Faculty at the Computation Laboratory where she continued her work on the Mark II and Mark III. Operators traced an error in the Mark II to a moth trapped in a relay, coining the term bug. This bug was carefully removed and taped to the log book September 9th 1945. Stemming from the first bug, today we call errors or glitch's [sic] in a program a bug.

Well known bugs

- Y2K – date overflow
- Ariane 5 Flight 501 – conversion overflow
- MIM-104 Patriot bug – clock drift
- Therac-25 – multiple causes

The Problem

- Every programmer knows they should write tests for their code. Few do.
  - The universal response to "Why not?" is "I'm in too much of a hurry."
  - This quickly becomes a vicious cycle- the more pressure you feel, the fewer tests you write.
  - The fewer tests you write, the less productive you are and the less stable you code becomes.
  - The less productive and accurate you are, the more pressure you feel.

Kent Beck, Erich Gamma

Testing Terminology

- *Validation* : a process designed to increase our confidence that a program works as advertised.
- *Verification* : a formal or informal argument that a program works on all possible inputs.
- *Testing* : a process of running a program on a limited set of inputs and comparing the actual results with expected results.
- *Debugging* : a process designed to determine why a program is not working correctly.
- *Defensive programming* : the practice of writing a program in a way designed specifically to ease validation and debugging.

Designing test cases

- Exhaustive testing is usually impossible
  - A program with three inputs ranging from 1 to 1000 would take 1'000'000'000 test cases.
  - With a speed of 1 test per second, it would take 31 years.
- How to define a limited set of good test cases ?
  - Black-box testing : testing from specification without regarding implementation or internal structure.
  - Glass-box testing : augments black-box testing by looking atimplementation.

Black-box testing

- Advantages:
  - not influenced by assumptions about implementation details
  - robust with respect to changes in implementation
  - allows observers with no internal knowledge of the program to interpret the results of the test
- Disadvantages:
  - unlikely to test all parts of a program

Glass-box testing

- Glass-box tests complement Black-box testing by adding a test for each possible path through the program's implementation.
  - A glass-box test set should be path-complete.

Golden Rule

Test for Success
Test for Failure
Test for Sanity

Test for Success

- The most fundamental part of unit testing is constructing individual test cases. A test case answers a single question about the code it is testing.
- In the case of Test for Success, a test cases shows that a feature works with a particular set of input.
- A test case should be able to
  - run completely by itself, without any human input. Unit testing is about automation.
  - determine by itself whether the function it is testing has passed or failed, without a human interpreting the results.
  - run in isolation, separate from any other test cases (even if they test the same functions). Each test case is an island.

Testing boundary conditions

- A program should test typical input values:
  - Arrays or sets are not empty.
  - Integers are between smallest and largest values.
- Boundary conditions usually reveal:
  - Logical errors where the path to a special case is absent.
  - Conditions which cause the underlying hardware or system to raise an exception (e.g. arithmetic overflow).
- Test data should cover all combinations of largest and smallest values:
  - Epsilon close to 0.001 and 0.00001
  - Arrays of 0 and 1 element
  - Empty strings and strings of one character

Test for Failure

- It is not enough to test that functions succeed when given good input; you must also test that they fail when given bad input.
- And not just any sort of failure; they must fail in the way you expect.
    - 1 + “a” = exception
    - 1 / 0 = exception
- What to do in case of failure?
    - Depends on the specification and the target language.
    - Exception, invalid return value, global error number, etc.

Test for Sanity

- You will find that a unit of code contains a set of reciprocal functions (ex: add/substract).
    - (1 + 1) + -1 = 1
    - (2 + 1) = (1 + 2) = 3
    - (5 * 5) * 0.2 = 5
    - decrypt(crypt(“cool”, key), key) = “cool”
- It is useful to create a “sanity check” to make sure that you can convert A to B and back to A without
    - losing precision,
    - incurring rounding errors
    - etc.

Black-box Example

- toRoman should return the Roman numeral representation for all integers 1 to 3999.
- toRoman should fail when given an integer outside the range 1 to 3999.
- toRoman should fail when given a non-integer number.
- fromRoman should take a valid Roman numeral and return the number that it represents.
- fromRoman should fail when given an invalid Roman numeral.
- If you take a number, convert it to Roman numerals, then convert that back to a number, you should end up with the number you started with. So fromRoman(toRoman(n)) == n for all n in 1..3999.
- toRoman should always return a Roman numeral using uppercase letters.
- fromRoman should only accept uppercase Roman numerals (i.e. it should fail when given lowercase input).

Glass-box Example

```
if type(input) != type(1):
   raise TypeError, "expected integer, got %s" % type(input)
if not 0 < input < 4000:
   raise ValueError, "Argument must be between 1 and 3999"
ints = (1000, 900,  500, 400, 100,  90, 50,  40, 10,  9,   5,  4,   1)
nums = ('M',  'CM', 'D', 'CD','C', 'XC','L','XL','X','IX','V','IV','I')
result = ""
for i in range(len(ints)):
   count = int(input / ints[i])
   result += nums[i] * count
   input -= ints[i] * count
return result
```