CSc 120
Introduction to Computer Programming II

Adapted from slides by Dr. Saumya Debray

05: Testing
Why test?

• Mars Climate Orbiter
  – Purpose: to study the Martian climate and to serve as a relay for the Mars Polar Lander
  – Disaster: Bad trajectory caused it to disintegrate in the upper atmosphere of Mars
  – Why: Software bug - failure to convert English units to metric values (pound-seconds vs. newton-seconds) as specified in the contract
Why test?

• THERAC-25 Radiation Therapy
  - 1985 to 1987: two cancer patients at the East Texas Cancer Center in Tyler received fatal radiation overdose (a total of 6 accidents) – *massive overdose*
  - Why: Software bug - mishandled race condition (i.e., miscoordination between concurrent tasks)
Why test?

• London Ambulance Service
  – Purpose: automate many of the human-intensive processes of manual dispatch systems associated with ambulance services in the UK – functions: Call taking
  – Failure of the London Ambulance Service on 26 and 27 November 1992
    o Load increased, emergencies accumulated, system made incorrect allocations
Purpose of testing

• Every piece of software is written with some functionality in mind

• Testing aims to identify whether the program meets its intended functionality
  – "testing can only prove the presence of bugs, not their absence"
  – the more thoroughly your software is tested, the more confidence you can have about its correctness

  – "Test until fear turns into boredom." – Kent Beck (?)
Testing and test cases

"thoroughly" ≠ lots of test cases

def main():
    x = input()
    if x % 2 == 1:  # x is odd
        do_useful_computation()
    else:
        delete_all_files()
        send_rude_email_to_boss()
        crash_computer()

It’s not enough to simply have a lot of test cases. They have to “cover” the program adequately.

make sure you use at least fifty different test inputs

1, 3, 5, 7, 9, 11, 13, 15, 17, 19, …
Approaches to testing

Black-box testing

• Focuses only on functionality
  - does not look at how the code actually works

• Good for identifying missing features, misunderstandings of the problem spec

White-box testing

• Focuses on the code
  - examines the code to figure out what tests to use

• Good for identifying bugs and programming errors
black-box testing
Black-box testing: what to test?

• Based purely on the desired functionality
  – shouldn’t be influenced by the particular code you wrote (that’s white-box testing)

• Aspects to consider:
  – expected outcome
    o normal vs error
  – characterizing values
    o edge cases vs “regular” values
Black-box testing: Outcomes

• Choose tests for both *normal* and *error* behaviors
  – assumes that we know what the error situations are

• Desired program behavior:
  – on normal inputs: produce the expected behavior
  – on error inputs:
    o detect and indicate that an error occurred
    o then behave appropriately as required by the problem spec

• Passing a test:
  – the program *passes a test* if it shows the desired behavior for that test
Black-box testing: Values

• Edge cases:
  – at or near the end(s) of the range of a value the program is supposed to operate on
  – Examples:
    o “zero-related”: 0, [], empty string, empty file, ...
    o “one-related”: 1, −1, list with one element, file with one line, ...
    o (maybe) large values

• “Regular” values:
  – not edge cases
Example:

“Read a file with one number per line. Print the sum of the numbers that occur on odd-numbered lines.”

Sample input file:

```
9
4
8
2
3
```
Example

“Read a file with one number per line. Print the sum of the numbers that occur on odd-numbered lines.”

Testing for outcome:

<table>
<thead>
<tr>
<th>Normal behavior</th>
<th>Error behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>• no. of numbers = 1</td>
<td>• a line has non-numeric characters</td>
</tr>
<tr>
<td>− 0 adds</td>
<td>• empty line</td>
</tr>
<tr>
<td>• no. of numbers = 3</td>
<td>• more than one number on a line</td>
</tr>
<tr>
<td>− 1 add; 1 skip in-between</td>
<td></td>
</tr>
<tr>
<td>• no. of numbers = 4</td>
<td></td>
</tr>
<tr>
<td>− 1 add; 1 skip at end</td>
<td></td>
</tr>
<tr>
<td>• &gt; 4 numbers</td>
<td></td>
</tr>
<tr>
<td>− several add operations</td>
<td></td>
</tr>
</tbody>
</table>
Example

“Read a file with one number per line. Print the sum of the numbers that occur on odd-numbered lines.”

Kinds of cases for normal behavior:

<table>
<thead>
<tr>
<th>Edge cases</th>
<th>Regular cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>• empty file</td>
<td>• a file with several numbers, one per line</td>
</tr>
<tr>
<td>• file with one number</td>
<td></td>
</tr>
</tbody>
</table>

Kinds of cases for error behavior:

<table>
<thead>
<tr>
<th>Error cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>• a line has non-numeric characters</td>
</tr>
<tr>
<td>• empty line</td>
</tr>
<tr>
<td>• more than one number on a line</td>
</tr>
</tbody>
</table>
• In black-box testing, what does the tester know about the code being tested?

__________________________________________

• When black-box testing, what are the kinds of cases we should test?

  o __________
  o __________
  o __________
  o __________

• How does white-box testing differ from black-box testing?

__________________________________________
Consider this program specification:

Write a program that reads a (possibly empty) file containing only numbers (and whitespace) and prints out the difference between the smallest and largest numbers. An empty input file should generate no output.

Specify sequences of lines that exemplify each of the following:

a) two error cases

b) two edge cases

c) one normal case
white-box testing
White-box testing: what to test?

• Ideally, that every path through the code works correctly
  - but this can be prohibitively difficult and expensive

• Instead, what we often do is:
  - check that the individual pieces of the program work properly
  - use asserts of pre/postconditions to check that the pieces interact properly
Unit testing

• Tests individual units of code, e.g., functions, methods, or classes
  – e.g.: given specific test inputs, does the function behave correctly?
    o CloudCoder!
  – useful for making programmers focus on the exact behavior of the function being tested
    o e.g., preconditions, postconditions, invariants
  – helps find problems early

• Isolate a unit and validate its correctness
• Often automated, but can be done manually
Code coverage

• Code coverage refers to how much of the code is executed ("covered") by a set of tests
  − want to be at (or close to) 100%
  − coverage tools report which parts of the program were executed, and how much
    o e.g., Coverage.py

• Figuring out how to increase coverage often leads to testing edge cases
Unit testing: practical heuristics

• Check both normal and error behaviors

• edge-case inputs:
  - zero values (0, empty list/string/tuple/file, ...)
  - singleton values (1, list/string/tuple/file of length 1, ...)
  - large values

• if statements: make sure each outcome (True/False) is taken

• Loops: test 0, 1, >1 iterations
Unit testing: what to check?

• Not just “output is what we expect”
  – remember “accidental” success

• Check that invariants hold at key points
Unit testing: what to check?

• Not just “output is what we expect”
  – remember “accidental” success
• Check that invariants hold at key points
Unit testing: what to check?

• Check that invariants hold at key points
Unit testing: what to check?

• Check that invariants hold at key points

① Check that nothing breaks if the loop does not execute at all
Unit testing: what to check?

• Check that invariants hold at key points

① Check that nothing breaks if the loop does not execute at all

② Check that everything is initialized properly when the loop is first entered
Unit testing: what to check?

- Check that invariants hold at key points

① Check that nothing breaks if the loop does not execute at all

② Check that everything is initialized properly when the loop is first entered

③ Check that everything is OK after going around the loop
Unit testing: summary

• Test normal and error values, edge cases

• If statements: test all branches (if/elif/else)

•Loops: check invariants for:
  – 0 iterations
  – 1 iteration
  – >1 iteration

• Functions:
  – check return values
Example 1: buggy list-lookup

# lookup(string, lst) -- returns the position where the given string occurs in lst.

def lookup(string, lst):
    for i in range(len(lst)):
        if string == lst[i]:
            return i
Example 1: buggy list-lookup

# lookup(string, lst) -- returns the # position where the given string # occurs in lst.

def lookup(string, lst):
  for i in range(len(lst)):
    if string == lst[i]:
      return i

0, 1, >1 iterations ⇒ lists of length 0, 1, 2
Example 1: (buggy) list-lookup

# lookup(string, lst) -- returns the # position where the given string # occurs in lst.

def lookup(string, lst):
    for i in range(len(lst)):
        if string == lst[i]:
            return i

0, 1, >1 iterations ⇒ lists of length 0, 1, 2
both branches taken ⇒ string is at positions 0, 1
Example 1: (buggy) list-lookup

```python
# lookup(string, lst) -- returns the
# position where the given string
# occurs in lst.

def lookup(string, lst):
    for i in range(len(lst)):
        if string == lst[i]:
            return i
```

0, 1, >1 iterations ⇒ lists of length 0, 1, 2
both branches taken ⇒ string is at positions 0, 1
some possible test inputs:
('a', []), ('a', ['a']), ('a', ['b','a'])
Example 1: (buggy) list-lookup

# lookup(string, lst) -- returns the # position where the given string # occurs in lst.

def lookup(string, lst):
    for i in range(len(lst)):
        if string == lst[i]:
            return i

0, 1, >1 iterations ⇒ lists of length 0, 1, 2
both branches taken ⇒ string is at positions 0, 1

some possible test inputs:
('a', []), ('a', ['a']), ('a', ['b','a'])

Note: this will catch the no-return-value bug
Example 2: (buggy) average

```python
# average(lst) -- returns the
# average of the numbers in lst.

def average(lst):
    sum = 0
    for i in range(len(lst)):
        sum += lst[i]
    return sum/len(lst)
```
Example 2: (buggy) average

```python
# average(lst) -- returns the average of the numbers in lst.
def average(lst):
    sum = 0
    for i in range(len(lst)):
        sum += lst[i]
    return sum / len(lst)
```

0, 1, >1 iterations ⇒ lists of length 0, 1, 2
Example 2: (buggy) average

```python
# average(lst) -- returns the average of the numbers in lst.

def average(lst):
    sum = 0
    for i in range(len(lst)):
        sum += lst[i]
    return sum/len(lst)
```

0, 1, >1 iterations ⇒ lists of length 0, 1, 2

some possible test inputs:

[]  [17]  [5, 12]
Example 2: (buggy) average

# average(lst) -- returns the average of the numbers in lst.

def average(lst):
    sum = 0
    for i in range(len(lst)):
        sum += lst[i]
    return sum/len(lst)

Note: this will catch the divide-by-zero on empty list bug

0, 1, >1 iterations ⇒ lists of length 0, 1, 2

some possible test inputs:
[ ], [17], [5, 12]
EXERCISE

Write four unit tests for the function below:

# Returns a list consisting of the strings in wordlist
# that end with tail.
def words_ending_with(wordlist, tail):
    outlist = []
    for item in wordlist:
        if item.endswith(tail):
            outlist.append(item)
    return outlist
Testing strategy

• Test as a part of program development
  - try out small tests even when the code is only partially developed (i.e., lots of stubs)
    o helps catch problems at function boundaries, e.g., number and types of arguments
    o can help identify bugs in the design, e.g., missing pieces

• Start with tiny test inputs (work your way up to small, then medium, then large)
  - problems found on tiny inputs are usually easier to debug