CSc 120
Introduction to Computer Programming II

07: Testing
Steps in software development

1. Specification
2. Design
3. Coding
4. Testing
5. Debugging
6. Maintenance

release
testing
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" - Dijkstra
  – the more thoroughly your software is tested, the more confidence you can have about its correctness
“thoroughly” != lots of test cases

def main():
    x = input()
    if x % 2 == 1:
        do_work()
    else:
        delete_all_files()
        send_rude_email_to_boss()
        crash_computer()
“thoroughly” != lots of test cases

```python
def main():
    x = input()
    if x % 2 == 1:
        do_work()
    else:
        delete_all_files()
        send_rude_email_to_boss()
        crash_computer()
```

It isn’t enough to simply have a lot of test cases. They have to “cover” the program adequately.
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
    ○ normal vs error
  – nature of values used
    ○ corner 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:
    ○ detect and indicate that an error occurred
    ○ 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

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

• “Regular” values:
  – not corner cases
Example  “Read a file containing integers and print the sum of the numbers at odd-numbered positions”

Testing for outcome (legal vs. error):

<table>
<thead>
<tr>
<th>Normal behavior</th>
<th>Error behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>• no. of numbers = 1</td>
<td></td>
</tr>
<tr>
<td>− 0 adds</td>
<td></td>
</tr>
<tr>
<td>• no. of numbers = 3</td>
<td></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>
<tr>
<td>• input file does not exist (or is unreadable)</td>
<td></td>
</tr>
<tr>
<td>• file has non-numeric characters</td>
<td></td>
</tr>
</tbody>
</table>
**Example**

“Read a file containing integers and print the sum of the numbers at odd-numbered positions”

**Testing for values (edge cases vs. regular values):**

<table>
<thead>
<tr>
<th>Corner cases</th>
<th>Regular values</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>• a file with several numbers over multiple lines</td>
</tr>
</tbody>
</table>
Example

“Read a file containing integers and print the sum of the numbers at odd-numbered positions”

Putting these together:

<table>
<thead>
<tr>
<th>Normal behavior</th>
<th>Error behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>• empty file</td>
<td>• nonexistent/unreadable file</td>
</tr>
<tr>
<td>• one number</td>
<td>• file has non-numeric characters</td>
</tr>
<tr>
<td>• 3 and 4 numbers</td>
<td></td>
</tr>
<tr>
<td>─ on the same line</td>
<td></td>
</tr>
<tr>
<td>─ on different lines</td>
<td></td>
</tr>
<tr>
<td>• &gt; 4 numbers</td>
<td></td>
</tr>
</tbody>
</table>

these can be combined in the same input file
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?
    ○ Cloud-coder!
  – useful for making programmers focus on the exact behavior of the function being tested
    ○ e.g., preconditions, postconditions, invariants
  – helps find problems early

• 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
    ○ e.g., Coverage.py

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

• Check both normal and error behaviors
• Corner-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, corner 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-look up

# 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

```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'])

**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 $\Rightarrow$ 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]
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)
    ○ helps catch problems at function boundaries, e.g., number and types of arguments
    ○ 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