Testing

CSC148, Introduction to Computer Science
Fall 2017
Diane Horton
A beginner’s way to test a function

• Write calls in the shell.
• Read the results and judge whether correct.
• What are the disadvantages of this?
Using doctest

• We have been giving examples in our docstrings.
• A module called doctest lets us run these!
• To run all the doctest examples in your module:

```python
if __name__ == '__main__':
    import doctest
doctest.testmod()
```
Thorough testing

• We’ll need more test cases.
• With doctest, our docstrings would be too long.
• With module pytest, code and its tests are in separate modules.
Choosing test cases

- Example: a function to find the maximum in a list.
- These test cases:

<table>
<thead>
<tr>
<th>List</th>
<th>Expected Result</th>
<th>Test passed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>[3, 6, 4, 42, 9]</td>
<td>42</td>
<td>yes</td>
</tr>
<tr>
<td>[22, 32, 59, 17, 18, 1]</td>
<td>59</td>
<td>yes</td>
</tr>
<tr>
<td>[1, 88, 17, 59, 33, 22]</td>
<td>88</td>
<td>yes</td>
</tr>
<tr>
<td>[1, 3, 5, 7, 9, 1, 3, 5, 7]</td>
<td>9</td>
<td>yes</td>
</tr>
<tr>
<td>[7, 5, 3, 1, 9, 7, 5, 3, 1]</td>
<td>9</td>
<td>yes</td>
</tr>
<tr>
<td>[561, 1024, 13, 79, 97, 4]</td>
<td>1024</td>
<td>yes</td>
</tr>
<tr>
<td>[9, 6, 7, 11, 5]</td>
<td>11</td>
<td>yes</td>
</tr>
</tbody>
</table>
Example

• Are you confident the function works?
• What if I said it passed 20 more test cases?
• What if I said it passed 100 more test cases?

• I hope you are skeptical.
• Quantity of test cases means little.
• *Quality* matters.
(In reality there are many more possible test cases)
Are we convinced?

- We want to believe the function works on *every possible input*.
- But the test cases seem to be chosen at random.
- We have no reason to believe the function works in the untested cases.
- And we can’t test them all!
A better approach

• Carve the possible inputs into meaningful categories.
• Pick a representative from each category to test.
• If we choose the category well
  – it will be reasonable to extrapolate from one test case to all other test cases in the category.
tested

untested

(In reality there are many more possible test cases)
tested

untested but reasonable to extrapolate
How to define the categories

• We base the categories on relevant properties of the inputs.

• Examples:
  – Size of something: 0, 1, larger
  – Size: even, odd
  – Position of something: beginning, ending, elsewhere
  – Relative location of two things: adjacent, separated
  – Presence of duplicates: yes, no
  – Ordering: unsorted, non-decreasing, non-increasing
  – Values of an integer: all negative, all positive, mixed
  – Etc.!!!
But which properties?

• We have to decide on which properties are relevant – there are so many.
• We decide based on knowing what the function or method does.
• If we know *how* it does it, that can influence our choices also.
  – For example, if a method divides a list in half, odd vs even size is pretty important!
def insert_after(lst: List[int], n1: int, n2: int) -> None:

"""After each occurrence of <n1> in <lst>, insert <n2>.
>>> lst = [5, 1, 2, 1, 6]
>>> insert_after(lst, 1, 99)
>>> lst
[5, 1, 99, 2, 1, 99, 6]
"""
Property-based testing

- Devising specific test cases is time-consuming.
- Property-based testing is an alternative:

<table>
<thead>
<tr>
<th>Instead of specifying this:</th>
<th>We specify this:</th>
</tr>
</thead>
<tbody>
<tr>
<td>A specific input, e.g., [3, 6, 4, 42, 9]</td>
<td>A category of input, e.g., lists of integers</td>
</tr>
<tr>
<td>A specific output, e.g., 42</td>
<td>A property of the output, e.g., returns an element</td>
</tr>
<tr>
<td></td>
<td>of the list or None</td>
</tr>
</tbody>
</table>

- Inputs (many!) are generated and tested automatically.
Thoughts on testing

• Designing test cases before writing code is a best practice in industry.
• It is part of test-driven development.
• When you test code, you must try to break it.
Fixing a bug

• When your testing reveals a bug, what to do?
• Beginners often:
  – Try some “typical” changes, e.g., change “>” to “>=“.
  – Add print statements.
• A rarely done but better strategy:
  – Trace the code on paper.
  – Why is this better?
• A professional strategy:
  – Use the debugger to trace it for you.
  – Use what you learn to hypothesize a fix.
Checking your fix

• Reap the benefit of having defined a thorough test suite with unittest:

Check your fix with the press of a button!
Professionalism

• We have seen two practices that are expected of any professional:
  – Test-driven development.
  – Using a debugger to find and fix bugs.

• You will hone these skills throughout the course.

• Professionalism is a theme we will revisit.