CPSC 301: Computing in the Life Sciences
Lecture Notes 14: Debugging

pre-class slides

Jessica Dawson
jqdawson@cs.ubc.ca
http://www.cs.ubc.ca/~jqdawson

University of British Columbia
Department of Computer Science

2016 W2
After completing this unit you will be able to:

• Develop a hypothesis-prediction-experiment strategy to locate and repair bugs in Python code
• Use the debugging facilities in the Spyder IDE to aid this process
“Bugs” and “Debugging”

- Errors in software are called “bugs”
  - Were described as far back as 1842 by Lady Augusta Ada King who was trying to program a mechanical computer designed by Charles Babbage
  - Term has been used to describe defects in engineering for over a century
  - It is often incorrectly attributed to Admiral Grace Hopper who was working on the early computer inside of which the following moth was found

- Removing errors is called debugging
Debugging: The Hard Truth

• We always write code that has bugs
  – Finding bugs is one of the most frustrating tasks in programming
  – Once you find the bug, fixing it is often easier
• There are good and bad ways to approach debugging
• Bad
  – Randomly change parts of your code
  – Ask the TA “Why won’t this work?”
  – Run the code again and hope that it works
• Good
  – A planned process of controlled scientific exploration
  – Ask your partner “Can you trace through this chunk of code?”
  – Save everything and take a break
Is There A Bug?

• If a bug is an incorrect program behaviour, what is correct?
• Before you write your program, you should create a specification: What does the program do?
• At the high level, the specification typically talks about some manipulation of data
  – For example, input a DNA sequence and output counts for each base
• As we already mentioned, together with the specification we should include test cases that are known to be correct and can be used to test whether the program is incorrect
  – For example, AACAGGTT will report 3 A's, 1 C, 2 G's and 1 T
• When the program behaves in a way that disagrees with its specification, then it has a bug
• Creating this specification and concrete test cases is the purpose of the function design recipe
Debugging in Python: Three Classes of Bugs

• Syntax errors
  – Bugs that Python spots before execution
  – Spyder will often spot the bug before execution, and flags it with red or yellow (if it is not sure) triangles
  – When executed the program will generate an error message immediately
  – The error message is often not very informative (eg: “invalid syntax”) but it does contain the line number where Python detected the error
  – The actual cause of the bug often occurs in the line above the one where Python detected it (or even several lines above)
Debugging in Python: Three Classes of Bugs

• Runtime errors:
  – Bugs that are spotted by Python during execution
  – Often called “exceptions”
  – An error message will be shown when Python detects the problem
  – The error message can be slightly more informative (eg: “ZeroDivisionError: integer division or modulo by zero”), and also contains the line number where the error was detected
  – These errors often involve applying an operation to a data type which does not support that operation
    • If the operation is wrong, the bug is easy to find
    • If the data is wrong, the bug is often much earlier in the program (the place where the wrong data was stored)
Debugging in Python: Three Classes of Bugs

• Semantic errors:
  – Bugs that Python cannot detect
  – The Python code executes, but the result does not satisfy the specification

• I would personally classify a program which gets stuck or enters an infinite loop as a program with a semantic error
  – It failed to satisfy the specification “program must finish”

• Some books also classify these behaviours as runtime errors
The Scientific Debugging Method

• Step 1: Identify a failure
  – A specific disagreement between program behaviour and specification (what the program is supposed to do)
  – Is the failure repeatable and consistent?

• Step 2: Form a hypothesis
  – Identify a potential defect in the program and explain why it would lead to the observed behaviour

• Step 3: Make a prediction
  – Describe a modification and what you expect to happen after that modification

• Steps 4–6: Perform an experiment, observe the results and draw a conclusion
  – Make the modification. Keep it simple—modify only one thing
  – If the observation does not match the prediction, you need to revisit your hypothesis and/or prediction (and undo your modification)

list adapted from http://cseweb.ucsd.edu/classes/su09/cse15L/method.php
Debugging in Spyder

• All IDEs include a **debugger**
  – Use the “Debug” buttons instead of “Run”

• Breakpoints and stepping
  – You can ask the debugger to stop execution just before any given command using a **breakpoint**
  – After stopping execution, you can continue one line at a time (**stepping**) or just resume execution as normal (**continue**)
  – When a line is a function call, you can **step inside** or **step over**

```python
# Examine the first character. Remember that P:
if first_line[0] == description_line_start:
    # The file appears to be a valid FASTA file.
    # The file appears to be a valid FASTA file.
    description_line so that the user knows w!
    # analyzed.
    print "Reading sequence: " + first_line[1:]
else:
```

70  # Examine the first character. Remember that P:
71  if first_line[0] == description_line_start:
72      # The file appears to be a valid FASTA file.
73      # The file appears to be a valid FASTA file.
74      description_line so that the user knows w!
75      # analyzed.
76      print "Reading sequence: " + first_line[1:]
77  else:
Why are you only tell me about the debugger now!? 

- The **debugger** is not a silver bullet
- Learning how to to trace code and predict what will happen on your own is a fundamental skill
  - Without it, it’s hard to understand the results when you use the debugger
- Relying too heavily on the debugger:
  - can detract from thinking through the specification and test cases *before* you start coding
  - Impede understanding what your code actually does and why it’s different from what you expect
  - Can be slow and cumbersome; stepping through yourself is often faster

“The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.”

- Brian Kernighan, 1979
Finding and Fixing Bugs in Python

• Class of bug is irrelevant to the task of finding and fixing it
  – Use the scientific method for debugging in every case
  – Syntax and runtime error messages give you a hint about where the bug might be, but the hint may not be correct

• Your IDE is trying to help you
  – Pay attention to flags in the editor (and other color coding)
  – Use breakpoints and step commands to slow down execution so that you can see what is happening
  – Use `print()` and variable explorer window to examine the values of variables
  – Keep a backup of your file