Lab: Basic Debugging Techniques

Overview

In this lab, you will use provided code that needs to be debugged. You will have a main program, which works as intended, but calls functions from a separate module, that doesn't always work as intended. There are three functions included in this module. Syntax errors are easy to spot because they likely will cause the program to fail. Logical errors, on the other hand, will result in incorrect outputs that are not detected by Python, i.e. the program will still run. It is up to you to find these.

You have already done a bit of debugging in the Exams, as well as in your own code using Spyder. You will learn some more advanced techniques before moving on to debug the included program.

Lab Activities

- Exercise 1: Debugging in Spyder
- Exercise 2: Debug the debug.py functions

Exercise 1: Debugging in Spyder

Overview: Get used to Breakpoints and Stepping – two of the most useful tools for debugging.

1. Open the debug_lab.py file. Make sure debug.py is saved in the same directory.	• You can drag the file directly into Spyder's editor window, use the Open button, or right-click the file and select "Open with…" and navigate to Spyder.

2. Set a Breakpoint at line 24	• Double click in the left-hand margin of the editor window, near or on the line number.
	• You should see a red dot appear, this is a **breakpoint**.
	  • This allows the code to "halt" at the specified point so that you can continue at your own pace and investigate what the code is doing.
	• Examine the code contained in the section for Exercise 1.
	• Click the "variable explorer" tab in the upper-right window of Spyder – where you read the docstrings. Select all existing variables there by pressing Ctrl+A, then rightclick and remove them all.
	  • If none exist, just leave it as-is.

3. Start debugging using Spyder.	• Hit the **blue play/pause** button instead, to the right. This is the debug button. • Usually you hit the green "play" button in the toolbar to execute a program. This time, the program knows it is being debugged and will adhere to all of your

	breakpoints. • Check the console window. It should display lines 15, 16 and 17, with arrows pointing to 16. This indicates the debugger has stopped at line 16. It shows you the next piece of code to be executed (line 17) as well as the code previously executed (line 16). • Also notice that line 16 is highlighted by the debugger, to show that code is paused there. • When debugging in Spyder, the program will **always stop** at the top of the code before proceeding.
4. Continue the program code to your breakpoint	• Hit the blue fast-forward button(>>), next to the stop/square button, in the toolbar. • Hovering your mouse over the button describes what it does. • Since we have an input, the code has stopped there. Check the console and continue. • Now you should be stopped at your breakpoint on line 24, as indicated by highlighting in the editor and the console window. • Use the blue "step" button, located to the right of the play/pause debug button, to execute the highlighted line (24) and move to the next. • We have just assigned a value to the variable **tot**. Go back to the variable explorer. Notice our variable is here, along with its value. This is very useful for keeping track of what variables contain what values at any given point. • Keep the program paused at line 25

5. Assign a different value to tot	• Another great thing about debugging, we can change variable values or execute functions on them, or whatever, **while the program is paused**. • To test this, go to the console window, and at the ipdb> prompt, type: • tot = 10 • Hit enter, and check the variable explorer. Tot should have a new value. • Hit the step button twice, which should now execute the print function on line 26 – **tot**'s new value should be reflected here as well. • You can even print(tot) or **import math** and then **math.sqrt(tot)** if you desired! It executes code just like the interpreteter. • This is useful for attempting different values during execution – but long term fixes must be implemented in the actual code. • Continue stepping through the code. To exit the debugger, hit the Stop button (the blue square). If you'd like the code to just continue, use the >> button.

Exercise 2: Debug the debug.py functions

Overview: Now that you have the basics down, use them to solve some simple bugs!

1. Try running the program as-is	• Try running the program and see if it works, or where it fails. • If desired, you can delete or comment out all of Exercise 1 so that it doesn't interfere.
2. Set breakpoints at the lines that call functions from the debug module	• You should have some idea what function caused failure, set a breakpoint there. • Start the debugger as before. • Continue on until the line you set the breakpoint at.
3. Step in to the debug.py functions	• Once the code is paused, use the "Step in" button, located to the right of the "Step" button. • This behaves similar to the step button, except it takes you **inside** of the function located at the breakpoint! Now you can see the code contained within the module, exactly as it is being executed. • Use the variable explorer, and the normal step button, to try and locate the source of the error. • Using the techniques learned so far, continue debugging the rest of the program! • Hint: #Function 3 does **not work properly** but it may appear to. Check the logic!