Computer Science 1MD3
Lab 3: Basic testing and the Python debugger

Aaron Simpson

Work Habits

Over the course of programming any application you're going to want to test your code frequently. The sooner that you catch an error the sooner it can be solved, and if an error is left unfound then the problems that arise from it can be more severe than having to sift through your entire code body after you already thought the program was finished. Missing one erroneous piece of code can mean that all the code you write thereafter has been based on false logic and needs to be rewritten. This is especially critical to avoid in business applications (and assignments) with deadlines attached to them. Once you do manage to track down a problem like this, very frequently the problem turns out to have been caused by a lack of diligence, rather than a judgment error on your part. Something you missed rather than something you were wrong about. This is why it is especially useful to have a tool which helps automate and organize the testing process for you. Such tools are included in most modern development environments. IDLE is no exception, and in this tutorial we will be going through the basics of using the Python “debugger”.

Debug Control

When testing a small program you've written often the most logical thing to do is start at the top of the code and scan through line by line, pretending to read the code like the compiler or interpreter would and stopping to think at each statement. You have to do this manually in your head because when you actually run your program the computer finishes virtually instantly, not giving you a chance to read along with it and understand what it is actually doing. Even though syntax errors (like a missing symbol or a bad name) usually get caught and pointed out specifically by compilers, logical errors (like your program working, but not giving the answer you expected it to) can only be located by you. This is an example of where the debugger's most basic function is most useful.

Go to the “Debug” menu and click “Debugger”. A checkmark will appear next to it in the menu and a new window called “Debug Control” will be opened. Move the windows so that you can see both the Python Shell and Debug Control. There are several new features to get used to in the debugger, but you can safely ignore most of them for the moment. By default, the “Stack” and “Locals” checkboxes should be checked. Check the boxes marked “Source” and “Globals” as well for the first test. Go to the Shell and type in the command *4+5*. Once you hit Enter you should see output in this form:

```
<pyshell#0>:1


'bdb'.run(), line 366: exec cmd in globals, locals
>'__main__', line 1: 4+5


                                    Locals

None



                                    Globals
__builtins__ <module'__builtin__' (built-in)>
__doc__      None
__name__     '__main__'
```

It may look a bit overwhelming at the moment but once you pick through it you may find some things that make sense. “>'__main__', line 1: 4+5” for example shows exactly the command that was taken in. What's important about this is that the program has stopped before continuing. Before we can continue we'll have to click the “Go” or “Step” button along the top left corner of the Debug Control window. The buttons are described as follows:

- The “Go” button will tell your program to proceed from its current point. It will keep going after until it reaches the end or a specially set “break point”.
- The “Step” button will move to the next line and stop again, and is good for scanning slowly through a critical area of your program. If you have a function nested inside another function, Step will also move into executing that and then wait.
- The “Over” button is somewhat like a less meticulous Step. The current statement will be read completely without stopping inside nested functions.
- The “Out” button rushes to the end of the program without stopping again, but still reading all of the code.
- The “Quit” button is not for exiting the Debug Control itself. The system will remain in debug mode when you click it. What it actually does is cancel the execution of the current statement the debugger is reading.

Next we should define what each of the sub-windows mean:

- The white section is the “Stack” area. This shows a hierarchy of what the current command being executed is. If you were reading through a branching set of *if* statements and having to remember your spot in them, this is what it would be like. In a moment you'll see how this works.

- Locals is a box in which the current values of the local variables are displayed.
- Globals is the same except it contains the current Global variables.

The variable listing sections will still not show anything as we haven't assigned one yet. Easier than trying to explain it all further would be to try something. Close the Python debugger and Shell completely. Open the Shell again and create a new module with the following body:

```
def dec2bin2(n):
    if n == 0:
        return ""
    else:
        if n % 2 == 0:
            a = "0"
        else:
            a = "1"
        return dec2bin2(n / 2) + a
```

Run this module. Once it finishes and you are able to enter commands again, go to the Debug menu and open the debugger (you should see the notification inside the Shell that it has been turned on). Turn the Globals and Source options back on. Once you've done that, go back to the Shell and type the following:

```
dec2bin2(5)
```

Immediately the command you enter should appear in the stack window. It tells you that in the main body of the program on the first line, *dec2bin2* has been called with the parameter *5*. Click Step. Now in the stack there will also be the line *if n == 0:* listed on line 2. The more these *if* statements stack as they soon will the more confusing it may seem, but it will also make the program's thinking much more apparent. If you look down to Locals you should see the line *n 5* listed. *n* is not a variable like we have been creating in the interpreter which stays existent and is accessible from pretty much anywhere, it is temporarily created to hold the “argument” that dec2bin2 was being called on, in this case the number 5. That is what makes it a local variable.

The final exercise for this tutorial is to step through this program line by line and read the output the debugger gives. Don't continue each step unless you understand what the parts you've already been introduced to are saying. If you need help or would like another program to experiment on ask your TA.