PythonWin Debugging tool

Readings

Read Zandbergen 11.1 - 11.5 to get his tips for debugging. Then read 11.11 and *dog-ear the section on debugging* as a checklist for you to review any time you hit a problem in your code.

The code below attempts to calculate a factorial through a loop that increments the multiplier by 1 until it reaches the original integer. This is a valid approach since 1 * 2 * 3 * 4 * 5 would also yield 120.

```python
# This script calculates the factorial of a given
# integer, which is the product of the integer and
# all positive integers below it.

number = 5
multiplier = 1

while multiplier < number:
    number *= multiplier
    multiplier += 1

print number
```

Even if you can spot the error, follow along with the steps below to get a feel for the debugging process and the PythonWin Debugging toolbar.

1. Open PythonWin and copy the above code into a new script.
2. Save your script as *debugger_walkthrough.py*. You can optionally run the script, but you won't get a result and you may have to shut down PythonWin in order to get back to where you were.
3. Click View > Toolbars and ensure Debugging is checked. Many IDEs have debugging toolbars like this, and the tools they contain are pretty standard: a way to run the code, a way to set breakpoints, a way to step through the code line by line, and a way to watch the value of variables while stepping through the code. We'll cover each of these in the steps below.
4. Set your cursor on the first line (number = 5) and click the Toggle Breakpoint button . A breakpoint is a place where you want your code to stop running so you can examine it line by line using the debugger. Often you'll set a breakpoint deep in the middle of your script so you don't have to examine every single line of code. In this example, the script is very short so we're putting the breakpoint right at the beginning. The breakpoint is represented by a circle next to the line of code and this is common in other debuggers too.
5. Press the Go button . This runs your script up to the breakpoint. You now have a small yellow arrow indicating which line of the script you are about to run.
6. Click the Watch button . What's commonly known as a watch window appears. This will help you track what happens to your variables as you execute the code line by line. Before you run any more code, however, you need to tell the watch window which variables to track.

7. In the Expression column of the watch window, double-click <New Item> and type the name of your first variable "number" (omit the quotes). In the Value column you'll see "NameError: name 'number' is not defined." This makes sense because you haven't run the line of code yet that creates this variable.

8. Similar to the previous step, click <New Item> again and set up a watch for the "multiplier" variable. You should get the same error about the variable not being defined yet.

9. Click the Step button . This executes one line of your code. Notice in your watch window that the variable "number" now has a value of 5.

10. Click the Step button again. This time the "multiplier" variable has been assigned a value.

11. Click the Step button a few more times to cycle through the loop. Go slowly and use the watch window to understand the effect that each line has on the two variables.

12. Step through the loop until "multiplier" reaches a value of 10. It should be obvious at this point that the loop has not exited at the desired point. Our intent was for it to quit when "number" reached 120.

Can you spot the error now? The fact that the loop has failed to exit should draw your attention to the loop condition. The loop will only exit when "multiplier" is greater than or equal to "number." That is obviously never going to happen as "number" keeps getting bigger and bigger as it is multiplied each time through the loop.

In this example, the code contained a logical error. It re-used the variable for which we wanted to find the factorial (5) as a variable in the loop condition, without considering that the number would be repeatedly increased within the loop. Changing the loop condition to the following would cause the script to work:

```python
while multiplier < 5:
```

Even better than hard-coding the value 5 in this line would be to initialize a variable early and set it equal to the number whose factorial we want to find. The number could then get multiplied independent of the loop condition variable.

13. Close PythonWin and re-open to a new script. Paste in the code below and save the script as debugger_walkthrough2.py.

```python
# This script calculates the factorial of a given integer, which is the product of the integer and all positive integers below it.
number = 5
loopStop = number
multiplier = 1
```
while multiplier < loopStop:
    number *= multiplier
    multiplier += 1
print number

14. Display the **Debugging** toolbar and step through the loop a few times as you did above. Watch the values of the "number" and "multiplier" variables, but this time, also add a watch on the "loopStop" variable. This variable allows the loop condition to remain constant while "number" is multiplied. Indeed you should see "loopStop" remain fixed at 5 while "number" increases to 120.

15. Keep stepping until "number" reaches 120 and you reach the "print number" line. At this line, don't press the **Step** button; instead, just press **Go** to finish out the script. (You don't want to step through all the internal Python code required to print the variable.) At this point the value of "number" should be 120, which is 5 factorial. If you want, you can try substituting other integers as the "number" value to find their factorials.

In the above example you used the Debugging toolbar to find a logical error that had caused an endless loop in your code. Debugging tools are often your best resource for hunting down subtle errors in your code.

If you reach an internal Python function such as print while using the Debugger, the debugger will dive right into all the Python code needed to run the function. You'll know when this happens because you'll see one or more windows open with code that's difficult to understand. This is also the case sometimes when you run arcpy functions. The problem is compounded because this type of code tends to call other functions, which winds up opening many windows.

If you don't want to see all this code, you can try shortcutting around it by using the **Step Over** button to jump over a complex function or **Step Out** to get out of the function. If stepping over or through or out of all that code is too confusing, you can set another breakpoint one or two lines beyond the line with the function and just press the **Go** button again to run to that next breakpoint. When you press the Go button, the debugger doesn't stop until it hits the next breakpoint.