National University of Singapore
School of Computing
CS1010X: Programming Methodology
Semester II, 2019/2020

Debugging Exercises II

In the first session, we learnt how to find the source of an error by interpreting error messages and inserting `print` statments. This can sometimes be tedious. For larger and more complex programs, programmers make use of debuggers to identify and correct the errors efficiently. For now, we provide a simple exercise to acquaint you with the debugger in IDLE.

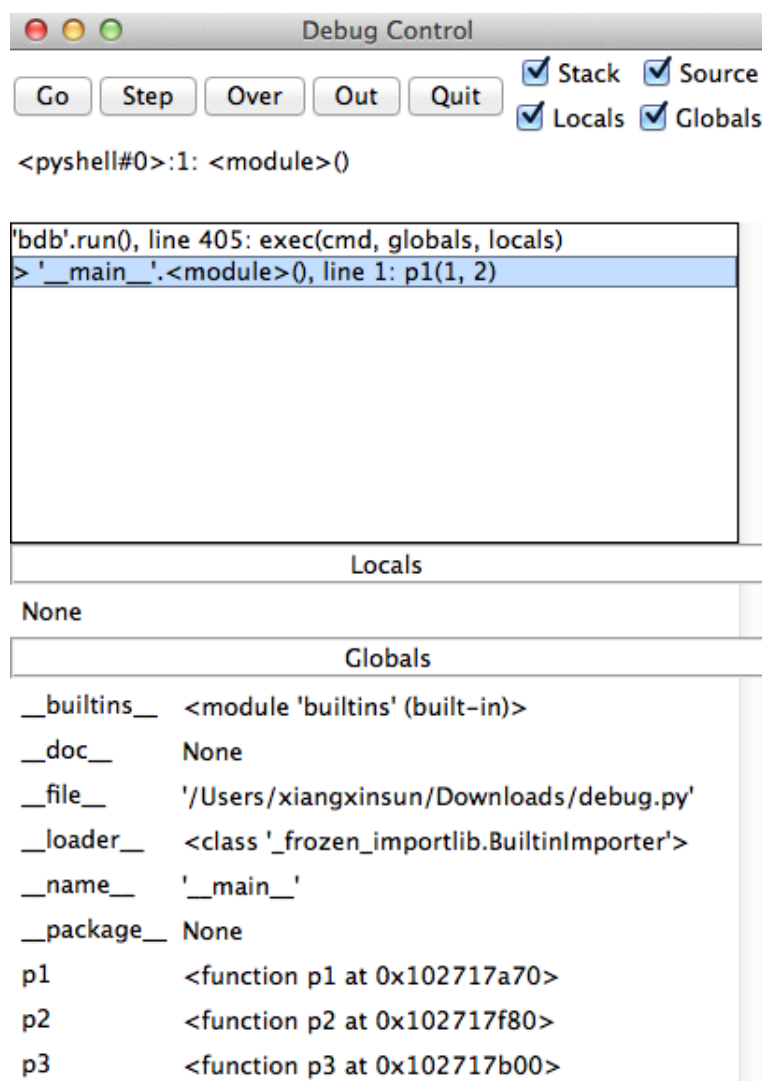### Stepping through evaluation in Debug Console

In this section we will continue using the same `debug.py` file as in the last one; however, we will step through the evaluation of `p1(1, 2)` in debug mode.

To start, perform the following steps:

1 Open `debug.py` file in IDLE and run it by pressing F5.

2 Click on Python Shell and make it the current focused window.

3 Select Debug > Debugger to reveal the debug console. You should see a new window Debug Control and a [DEBUG ON] sign in your Python shell.

4 Check all four options: Stack, Source, Locals and Globals. Now, instead of evaluating the expression to completion, we will evaluate the expression step by step.

5 Enter the following into the Python Shell again:

```
p1(1, 2)
```

You should see the following in the Debug Control window. Stretch and vertically en-large the Debug Control window if you are unable to see the 'Locals' and the 'Globals'.
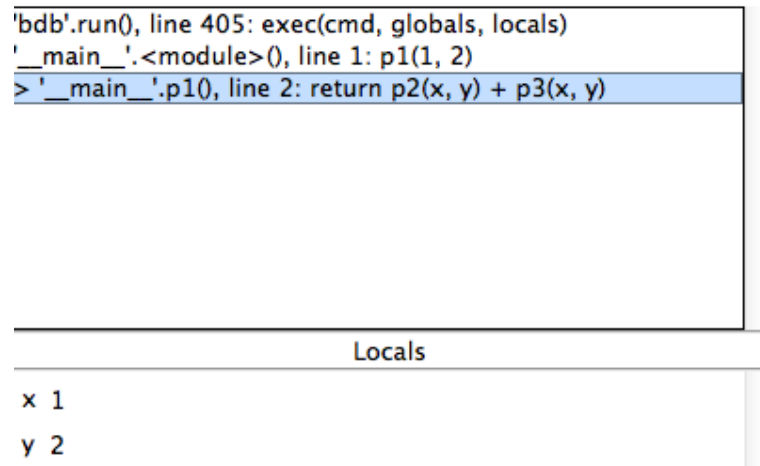
There are several sections in the debug console. It has several buttons at the top: Go, Step, Over, Out and Quit. For now, we are only interested in the Step button, which allows you to evaluate one step at a time. The large box on top shows you the call stack. The Locals tab shows local variables and their values if any. The Global tab shows global variables and their values if any. Currently we can see 3 familiar global variables p1, p2 and p3 which are defined as functions.

The line highlighted in the source code box is the next statement to be executed. Currently it is

```
> '__main__'.<module>(), line 1: p1(1, 2)
```

You may ignore the part that doesn't look familiar to you and focus on the useful part line1: p1(1, 2). That says that the interpreter is current paused right before executing p1(1, 2).
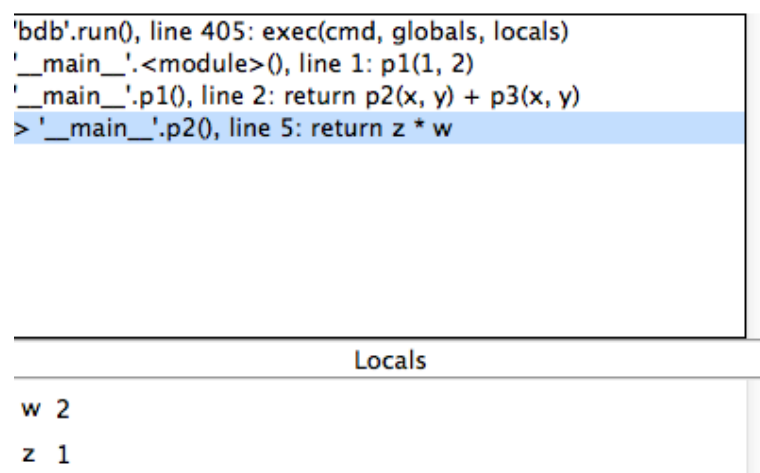
Click on Step button. You will notice that several things have changed:

```
'bdb'.run(), line 405: exec(cmd, globals, locals)
'__main__'.<module>(), line 1: p1(1, 2)
> '__main__'.p1(), line 2: return p2(x, y) + p3(x, y)




                         Locals
 x 1
 y 2
```

Now the interpreter has entered function p1. There are two local variables: x of value 1 and y of value 2. The next line to execute becomes `return p2(x, y) + p3(x, y)`. Observe that the statement should first invoke p2 and then p3.

Click on Step again and you should see:

```
'bdb'.run(), line 405: exec(cmd, globals, locals)
'__main__'.<module>(), line 1: p1(1, 2)
'__main__'.p1(), line 2: return p2(x, y) + p3(x, y)
> '__main__'.p2(), line 5: return z * w



                         Locals
 w 2
 z 1
```

You can probably figure out what's going on without our explanation now! The interpreter has entered function p2. There are two local variables, w of value 2 and z of value 1. The next line to be executed is `return z * w`.

Click on Step again.

```
'bdb'.run(), line 405: exec(cmd, globals, locals)
'__main__'.<module>(), line 1: p1(1, 2)
'__main__'.p1(), line 2: return p2(x, y) + p3(x, y)
> '__main__'.p3(), line 8: return p2(a) + p2(b)




                              Locals
a  1
b  2
```

The interpreter has completed the evaluation of p2 and has proceeded on to deal with the call to p3.

Click on Step again. Now, you will see a TypeError showing up. This tells you that the error described earlier has occurred when running this statement. Specifically, the error was encountered when evaluating the first part of this statement p2(a).

```
debug.py:8: p3()
TypeError: p2() missing 1 required positional argument: 'w'
'bdb'.run(), line 405: exec(cmd, globals, locals)
'__main__'.<module>(), line 1: p1(1, 2)
'__main__'.p1(), line 2: return p2(x, y) + p3(x, y)
> '__main__'.p3(), line 8: return p2(a) + p2(b)
```

Now that you have discovered the error, click Quit to abort the current code execution.

Other buttons may help speed up the debugging process by skipping one or more lines.

Go: Clicking this will run the program until the next break point is reached. You can insert break points in your code by right clicking and selecting Set Breakpoint. Lines that have break points set on them will be highlighted in yellow.

Step: This executes the next statement. If the statement is a function call, it will enter the function and stop at the first line.

Over: This executes the next statement just as Step does. But it does not enter into functions. Instead, it finishes executing any function in the statement and stops at the next statement in the same scope.

Out: This exits the current function and stops in the caller of the current function. After using Step to step into a function, you can use Out to quickly execute all the statements in the function and get back out to the outer function.

Quit: This terminates execution.

You may want to try out these buttons using debug.py and see what happens.

Now you may proceed to training questions.