Program 1

Put this code in a new Python file, but do not run it.

```
lst = [[1, 2], [3, 4], [5, 6], [7, 8]]

def magic(x):
    s = 0
    for y in x:
        z = y[0]
        s += z
    return s

if __name__ == '__main__':
    print(magic(lst))
```

What do you expect this program to do? What exactly will it print? (If you cannot answer either or both of these questions easily, you should strongly consider reviewing lecture slides until you can - but debugging may help you understand it)

Now, run the program. Does it do what you expected? If not, we can use a debugger to reconcile this difference. If it does do what you expected, you will still use a debugger to learn how to use debuggers.

Setting a breakpoint

A debugger will start by running your program. However, eventually your program will either encounter an error or hit a breakpoint - at which point the debugger will pause the program and let you examine it.

Set a breakpoint on the `z = y[0]` line. You can do this by moving your mouse cursor to the left of the line numbers, and clicking on the red dot that appears. You can click on the red dot again to remove an existing breakpoint.

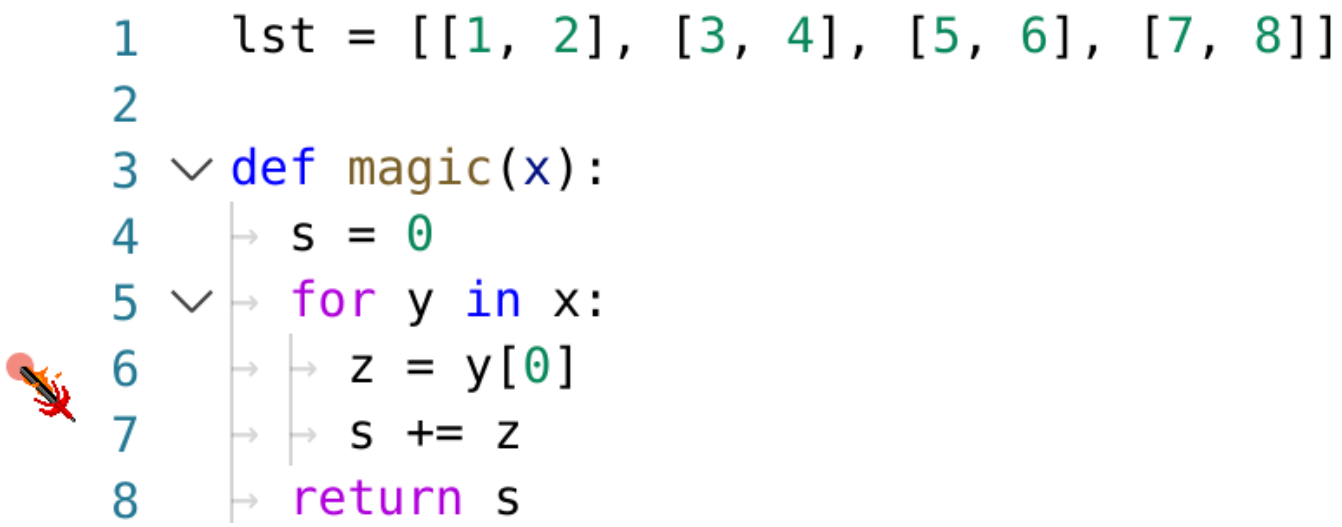

Figure 1: breakpoint.png

Debugging

Once you have set your breakpoint, select the 'Start debugging' option in the 'Debug' menu in the menubar (usually at the top of the program window or at the top of your screen - where the File, Edit, etc. menus are). If you are asked to choose a debug configuration, select 'Python File.'

Visual Studio Code will switch into debugging view. Four panels (Locals, Watch, Call stack, and Breakpoints) will appear to the left of your Python file, and a floating debugging controller bar will appear at the top of the window.

When a breakpoint or error is reached, the program will pause and the line that was being executed will get highlighted.

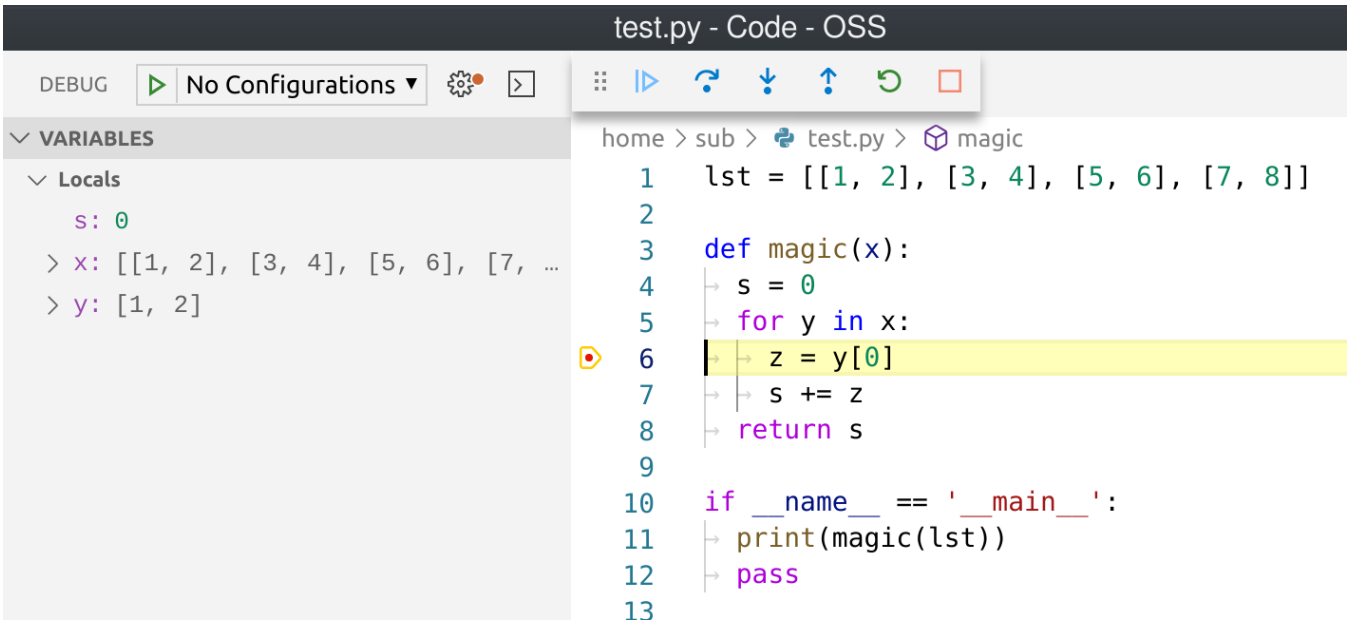

Figure 2: broke.png

Observe the Locals window. This lets you inspect the values of all the local variables (in this case, all variables inside the magic function). You don't need to do it right now, but you can edit variable values inside the locals window by double clicking - this can be useful for testing how your code reacts to different situations.

Controls

Using the debug control bar, start stepping through the code. Observe how the variable values are updated in the locals window - they will be highlighted when the value changes.

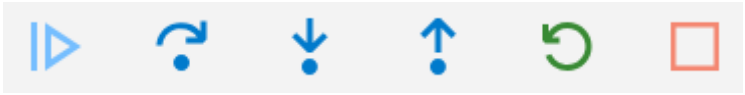

Figure 3: controlbar.png

Here is a list of what the control bar buttons do, from left to right:

- Run: continue program execution until a breakpoint is hit or an error occurs
- Step Over (semicircle arrow over dot): execute the entire current line of code
- Step Into (down arrow): execute the current line of code, but if it calls a function then go into that function
- Step Out (up arrow): finish executing the current function, then pause
- Restart debugging: if you make changes to the code while debugging, you will need to restart debugging to allow the new code to be loaded
- Stop debugging: exit debugging completely

For now you should use Step Over.

Step through the code and observe what happens. If you don't fully understand what this program does, look at the locals window to see what the values of `y` and `z` and `s` are. Once you have done some stepping, you can Run or Stop Debugging.

Program 2

Load this code into your computer in a new file, and set a breakpoint on the `if` line:

```
q = input('Enter a number: ')
if q == 10:
    print('Your number was 10')
else:
    print('Your number was not 10')
```

Read the code & think about what you expect this program to do. Then run the program to verify that you were correct (you may need to run it more than once to verify this).

Now, start debugging & enter 10 as the input. The breakpoint on the 'if' line will get hit and you will be able to view the locals. The locals window will show some stuff that starts with '___'; you can ignore all of those - they are variables that Python loads into the file. The only one that is relevant here is 'q.'

The bug in this code is that it will always state that your number was not 10 - even if you enter 10 as the input. In other words, the if condition (`q == 10`) is always False, even though we want it to be True when the user enters 10 as the input. Click the '+' in the Watch window header and enter the following watch:

```
q == 10
```

The watch window can store many single lines of code, and will show you the result of evaluating them in the current context. In this case, we are asking Python to evaluate the code `q == 10`, and the watch window tells us that this evaluates to `False` - so the `else` case is triggered, since our `if` condition is also `q == 10`. Now you can look closely at the locals window and observe that q is in fact not `10`. Q is `'10'`: the string 10. And now you can fix the bug in this program.

Now go back to the previous file with the magic() function, debug it (keeping the same breakpoint at `z = y[0]`), and when it breaks enter this expression as a new watch:

```
sum(x)
```

The watch window can contain arbitrary expressions and will always show you their current value. The watches are updated as you step through the code, just like the locals.

Program 3

Load this code into your computer:

```
def product(lst):
    # multiplies the two numbers in a two-number list
    first = lst[0]
    second = lst[1]
    product = first * second
    return product

def recProductSum(lst, sum):
    if lst == []:
        return sum
    firstElem = lst[0]
    newSum = sum + product(firstElem)
    restOfList = lst[1:]
    return recProductSum(restOfList, newSum)

print(recProductSum([[1,2], [3,4], [5,6]], 0))
```

`recProductSum` is a recursive function that takes a list of number pairs and sums the products of these pairs.

Set a breakpoint on the first line inside the `recProductSum` function (the one with 'if').

The call stack window shows you the functions that have been called. When you first hit the breakpoint, this file will have run (module) and then the `recProductSum` function will have been called at the bottom of the file and then your breakpoint will have been hit. Thus, the call stack will be the file ran and then called the `recProductSum` function.

Now, Step Over the code until you get to the 'return' line. When you Step Over one more time, what do you expect to happen in the call stack?

The call stack will grow to have one more recProductSum. This is because the file (module) called recProductSum, and then you manually stepped to the return line at which point the function recursively calls recProductSum again - so now we are two deep inside recProductSum.

Now, step through the function again but use Step In (down arrow) instead of Step Over. Notice that when you get to the line that computes newSum, your program will step into the 'product' function instead of just skipping over it. Also notice that the locals window has switched to only showing variables that exist inside product - variables from `recProductSum` have disappared.

You can click on a line in the Call stack window to show variables from that level in the stack. This can be useful when writing recursive programs, since you can examine what the values of variables are at each depth of the recursive call. To do this easily, put a breakpoint *inside* the if statement for the base case. When the base case triggers, your program is at the max recursive depth, so you can examing the entire call stack right before it starts to unwind.

You (should) know that product is a correct function and so there's no need to debug it. Thus, you can use Step Out (up arrow) to run until the product function is over - at which point you will be paused back at the newSum line in `recProductSum` from which product was called.