COMP 1510 Programming Methods Lab 05

Christopher Thompson
chris_thompson@bcit.ca

BCIT CST — February 2022

Welcome!

Welcome to your fifth COMP 1510 lab.

For lab five, I'd like to take you on a formal tour of the PyCharm debugger.

The Python debugger gives us access to the same sorts of things you will use in other debuggers – breakpoints, steps, watches. We will use a common debugging vocabulary that you can apply to other languages and other debuggers.

Be prepared to take some screenshots to "prove" that you've completed each step.

Let's begin!

1 Grading

Figure 1: This lab is graded out of 5

This lab will be marked out of 5. For full marks this week, you must:

1. (5.0 points) Correctly implement the debugging requirements in this lab.
2. (-1.0 point penalty) I will withhold one mark if your commit messages are not clear and specific. Tell me EXACTLY what you did. You must:
   (a) Start your commit message with a verb in title case in imperative tense (just like docstrings). Implement/Test/Debug/Add/Remove/Rework/Update/Polish/Write/Refactor/Change/Move...
   (b) Remove unnecessary punctuation – do not end your message with a period
   (c) Limit the first line of the commit message to 50 characters
   (d) Sometimes you may want to write more. This is rare, but it happens. Think of docstrings. We try to describe a function in a single line, but sometimes we need more. If you have to do this, leave a blank line after your 50-char commit message, and then explain what further change(s) you have made and why you made it/them.
   (e) Do not assume the reviewer understands what the original problem was, ensure your commit message is self-explanatory
   (f) Do not assume your code is self-explanatory.

2 Submission Requirements

1. This lab is due before your tutorial begins on **Thursday February 10th 2022.**
2. Late submissions will not be accepted for any reason.
3. **This is a collaborative lab**. I strongly encourage you to share ideas and concepts. Work together. But please submit your own version to GitHub for your own marks. If I see evidence any clandestine black market for screenshots I will be very upset.

3 Set up

1. Visit this URL to copy the template I created to your personal GitHub account, and then clone your repository to your laptop:

   `https://classroom.github.com/a/fsizY0pk`

2. Populate the README.md with your information. Commit and push your change.

4 Style Requirements

1. You must observe the code style warnings produced by PyCharm. **You must style your code so that it does not generate any warnings.**

   When code is underlined in red, PyCharm is telling us there is an error. Mouse over the error to (hopefully!) learn more.

   When code is underlined in a different colour, we are being warned about something. We can click the warning triangle in the upper right hand corner of the editor window, or we can mouse over the warning to learn more.

2. **You must comment each function you implement with correctly formatted docstrings.** Include informative doctests **where necessary and possible** to demonstrate to the user how the function is meant to be used, and what the expected reaction will be when input is provided.
3. In Python, functions must be atomic. They must only do one thing. If a function does more than one logical thing, break it down into two or more functions that work together. Remember that helper functions may help more than one function, so we prefer to use generic names as much as possible. Don't hard code values, either. Make your functions as flexible as possible.
4. Don't create functions that just wrap around and rename an existing function, i.e., don't create a divide function that just divides two numbers because we already have an operator that does that called `/` .
5. **Ensure that the docstring for each function you write has the following components (in this order)**:

   (a) Short one-sentence description that begins with a verb in imperative tense and ends with a period.
   (b) One blank line
   (c) Additional comments if the single line description is not sufficient (this is a good place to describe in detail how the function is meant to be used)
   (d) One blank line if additional comments were added
   (e) PARAM statement for each parameter which describes what the user should pass to the function
   (f) PRECONDITION statement for each precondition which the user promises to meet before using the function

(g) POSTCONDITION statement for each postcondition which the function promises to meet if the precondition is met

(h) RETURN statement which describes what will be returned from the function if the preconditions are met

(i) One blank line

(j) And finally, the doctests. Here is an example:

```
def my_factorial(number):
    """Calculate factorial.

    A simple function that demonstrates good comment construction.

    :param number: a positive integer
    :precondition: number must be a positive integer equal to or
                   greater than zero
    :postcondition: calculates the correct factorial
    :return: factorial of number

    >>> my_factorial(0)
    1
    >>> my_factorial(1)
    1
    >>> my_factorial(5)
    120
    """
    if number == 0:
        return 1
    else:
        return number * factorial(number - 1)
```

5 Requirements

Please complete the following:

1. **Open the module named step_01.py**:

   (a) We can execute the module by choosing Run 'step_01' when right-clicking the source code in the code editor pane, or by choosing Run > Run from the main menu, etc.

   (b) We can debug the module by choosing Debug 'step_01' when right-clicking the source code in the code editor pane, or by choosing Debug > Run 'step_01' from the main menu, etc.

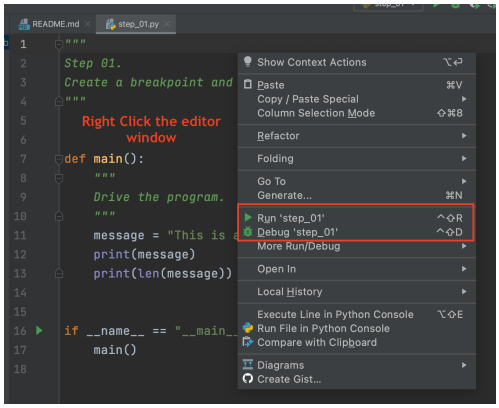

Figure 2: Right-click the editor pane to run or debug your code.

(c) **Debug the module**. Don't add a breakpoint. Just choose to debug it.

(d) When we debug a module, the Debug pane opens at the bottom of PyCharm. Since we did not set a breakpoint, the program ran to execution. The Console pane inside the Debug tab should look something like this:

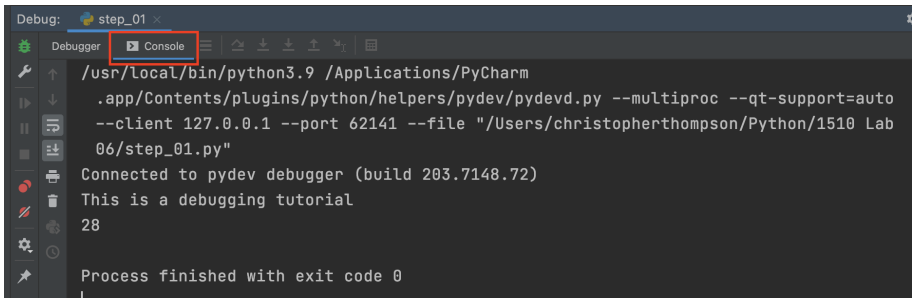

Figure 3: The Console tab in the Debugger pane shows us what happened.

(e) In order to use the debugger, we have to set a breakpoint first.

(f) **Set a breakpoint on line 11** in the module (where we declare and initialize the variable message) by left-clicking in the left-hand margin (sometimes called the gutter) of the editor pane, between the line number and the actual line of code. You will know you have set a breakpoint because it appears like a large red dot:

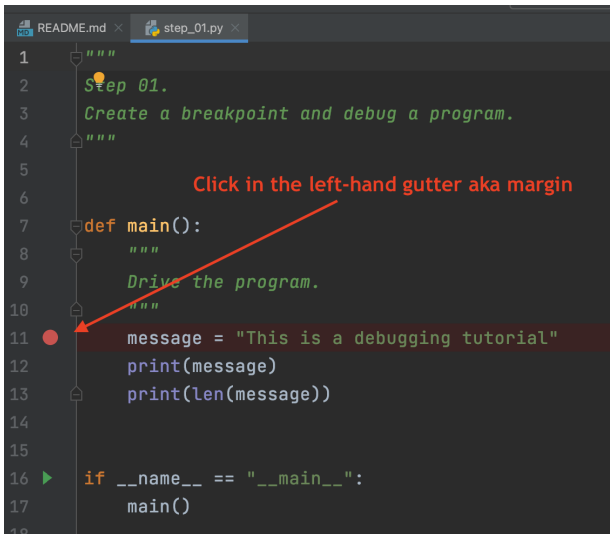

Figure 4: Click in the left-hand margin to add a breakpoint.

(g) If you carefully mouse over the breakpoint, you will see a little info window that pops up. It tells us that the breakpoint will suspend this thread. (We will talk about threads next term – they are part of concurrency and parallel programming. A thread is a sequence of instructions managed by an operating system's thread scheduler. We are writing single-threaded programs. Next term we will begin to explore threading aka concurrency and parallel programming. It's super fun and important and includes some neat concepts like semaphores, locks, races, etc.)

Figure 5: Mouse over the breakpoint to see some neat info about it.

(h) **Now debug the program**. Execution will begin and pause at the beginning of the line that contains the breakpoint. The line has not executed yet. Everything up to that point has executed, and the program suspends execution for us.

(i) Examine the debugging information that appears in the Debug pane. **Identify and explore** the following things:

i. The stack of function calls (this will contain <module> and main right now)

ii. The list of special variables (click on <module> in the stack to reveal this)

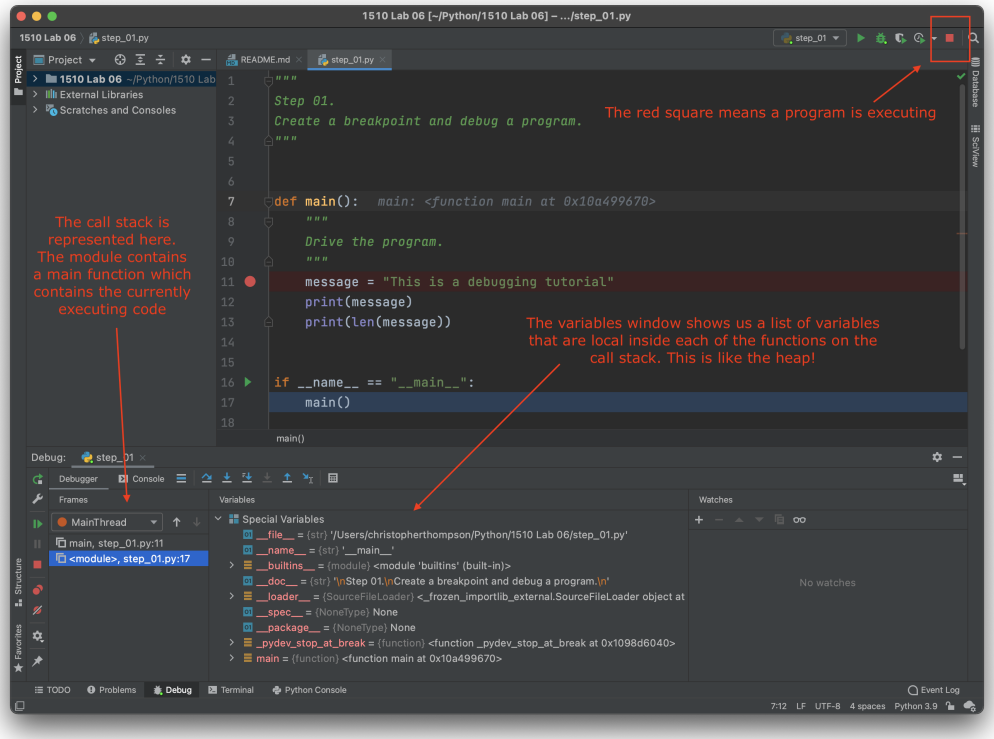

Figure 6: Here's what we see when we reach a breakpoint during debugging.

(j) So far so good?

(k) **Examine the buttons** above the stack of function calls in the Debugger pane. They are composed of blue arrows and white bars. The white bars are supposed to symbolize lines of code. The blue arrows symbolize what we want to do. Mouse over the buttons to identify the following:

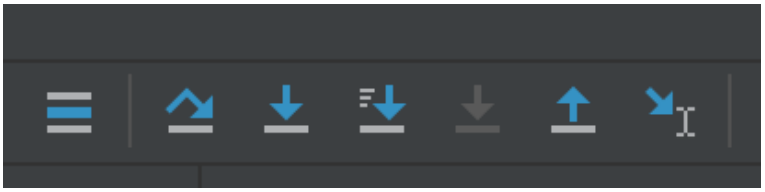

Figure 7: These buttons are what we use after we reach a breakpoint.

i. **Show Execution Point**: Highlights the current execution point in the editor and shows the correct function in the call stack.

ii. **Step Over**: Executes the program until the next line in the current function or module, without trying to debug any functions invoked in the current line. If we are at the last line of a function, the debugger moves the point of execution to the next correct line in the module.

iii. **Step Into**: The debugger steps into the function being invoked at the current execution point.

iv. **Step Into My Code**: Steps into the next line of your code, ignoring any calls to functions in outside libraries.

v. **Force Step Into**: Forces the debugger to step into the next function even if it's not one of yours.

vi. **Step Out**: Forces the debugger to finish the current function and return to the function that called it.

vii. **Run To Cursor**: Place your cursor somewhere in the editor and then click this. The debugger will execute your code until it reaches your cursor, pausing if it encounters any other breakpoints.

(l) **Click Step Into My Code**. This will execute the current line of code (11) and move to the next line (12) without executing it.

(m) **Observe** in the debugger window that when you click on the main function in the stack, a variable called message is now in the Variables list. That is because line 11 has executed and there is now a string in memory bound to the identifier message that has local scope in the main function:

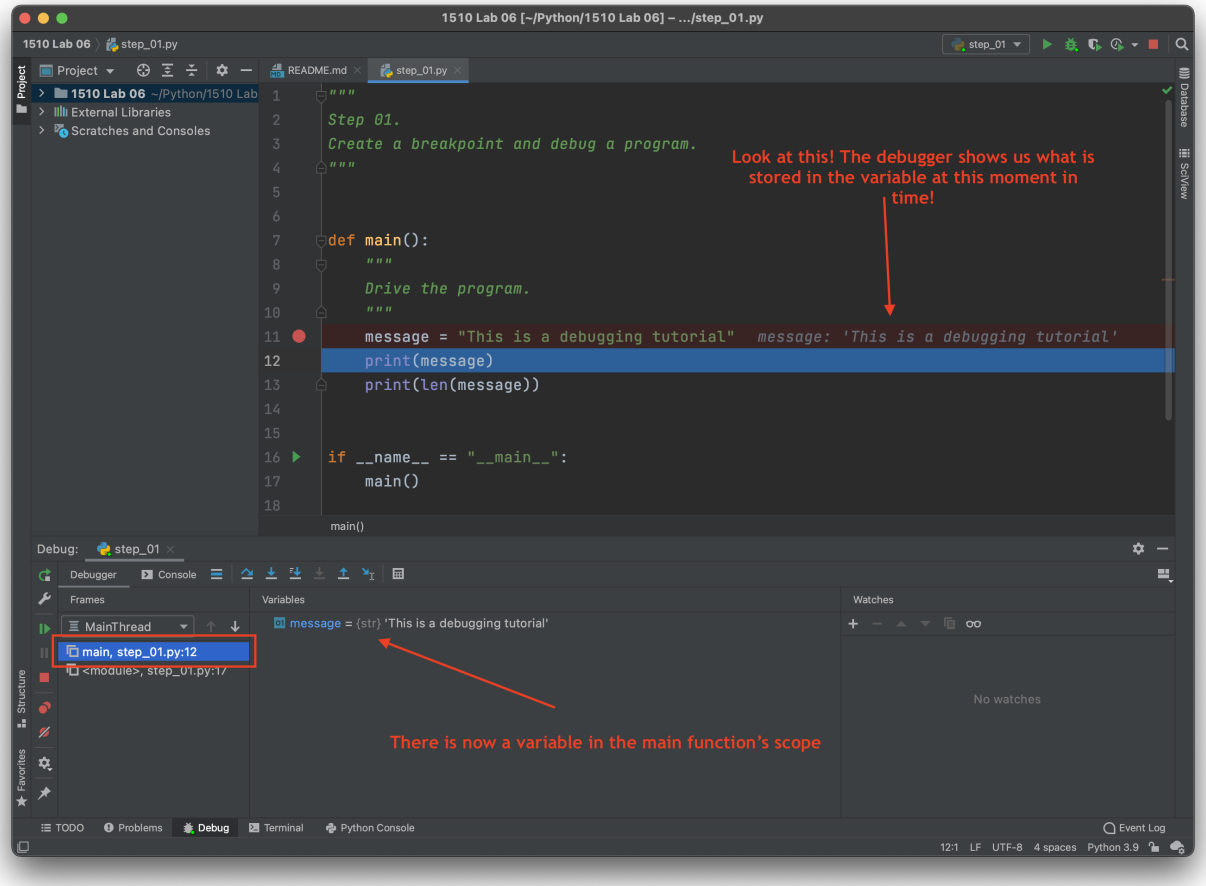

Figure 8: Here's what I saw after stepping into my code.

(n) **Examine** the editor pane, too. Look on line 11. What do you see?

(o) If you choose the <module> in the call stack, can you still see the Special Variables?

(p) **Press the button called Step Into My Code again**. What happened? Did it execute the next line of code, line 12?

(q) Yes it did. If you look carefully along the top of the Debugger pane, you will see it actually contains two sub-panes called Debugger and Console. **Click the Console sub-pane**. You will see the program output here: "This is a debugging tutorial"! Line 12 has executed.

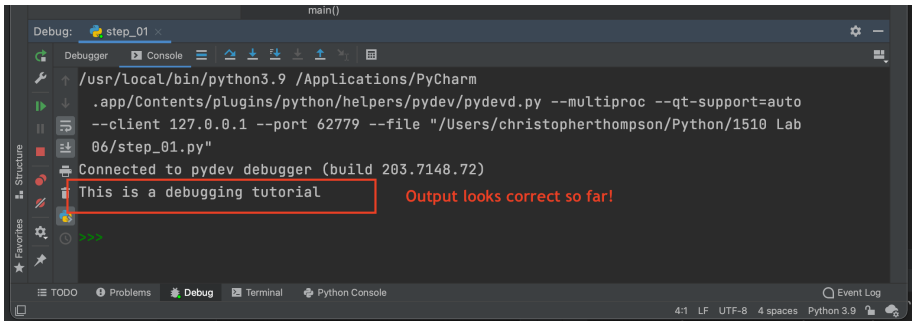

Figure 9: Looks good so far!

(r) **Return to the Debugger sub-pane** inside the Debugger pane. This time instead of Step Into My Code, I want you to press the button called Step Out. What happened? Is the program still running? Did the Debugger kick you out of the main function? Did it execute the final line of code in main( )?

(s) The debugger is still running. I can tell because of the red square on the left-hand side of the Debugger pane and in the upper right-hand corner of PyCharm. Anytime a program is running, that red square will be visible.

(t) The final line of code in main (line 13) also executed. If you view Debug Pane > Console you will see the correct output, i.e., the length of the string (28).

(u) Note that the main function has disappeared from the stack. It has executed, the flow of control has returned to line 17 where we called main in the if-statement at the bottom of the module.

(v) **Press Step Out again**. The program ends and debugging is complete.

(w) By the way, you can remove a breakpoint by clicking it. It's that easy!

2. **Now let's examine the module called step_02.py**:

(a) **Set a breakpoint on line 21** where message is declared and initialized.

(b) **Start the debugger**. It will execute the program up to but not including the line that contains the breakpoint.

(c) This time, **press Step Into (1)**. Observe the change in the Debugger panes.

(d) **Press Step Into again (2)**. We have entered the function called display_urgent_message. Observe the change in the function call stack, the Variables list, and the code editor window. Note the helpful comment added to the end of line 12.

(e) **Press Step Into again (3)**. This time, the flow of control has moved into the convert_to_upper_case function. Note the changes in the function stack and Variables list.

(f) **Press Step Into again (4)**. There are now two variables with local scope in the convert_to_upper_case function.

(g) If you **press Step Into yet again (5)**, the flow of control will return to the display_urgent_message function. Press Step Into one more time to execute line 13 so there are two local variables in the scope of display_urgent_message.

(h) How many more times must you press Step Into before the program ends?

(i) **Debug step_02.py again**. Start by pressing Step Into. When the debugger reaches line 13, instead of pressing Step Into, press Step Over. What happens?

(j) **Debug step_02.py one final time** but this time, instead of pressing Step Into, press Step Over. What happens differently each time you press it?

(k) **Modify the README.md**. In one short ELI5 (Explain it like I'm 5 years old) paragraph, tell me how Step Over and Step Into are different. Put this paragraph under a header called *** Step 2 ***

3. I'd like to introduce you to conditional breakpoints. **Open step_03.py**:

   (a) **Try to run the module**. It crashes! That's because we need to furnish a command line argument. Do you remember how to do this:

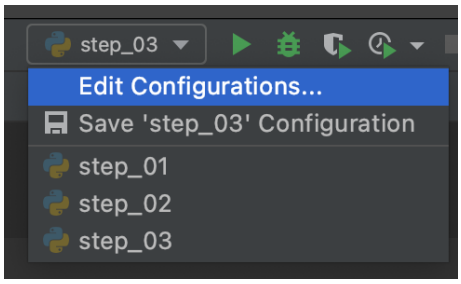

Figure 10: Choose Edit Configurations...!

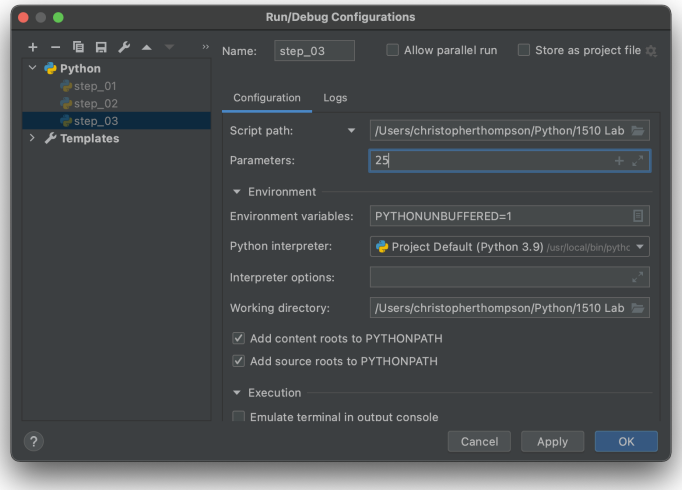

Figure 11: And add a command line argument (I chose 25).

   (b) The program should now execute sans problèmes and generate a solution: 6204484017733239439360000.

   (c) Suppose we want to **put a breakpoint on line 15** where the multiplication takes place. We want to see what the variables store in the middle or the end of the loop, i.e., when integer equals 10 or 24, etc.

   (d) Go ahead and put a breakpoint there. Debug the program. How many times do you have to click Step Over quite a few times before we reach the end of the loop because we start at the breakpoint during the first iteration.

   We can click Step Out, we still have to click several times before we reach the state we seek.

   But what if we wanted to ignore the breakpoint until integer contains the value 25?

   (e) **Let's make our breakpoint a "conditional" breakpoint**. The debugger ignores the breakpoint until the condition is true. Right-click the breakpoint to reveal the Breakpoints pane. In the Condition field, let's tell the debugger to ignore the breakpoint until integer is equal to 24:

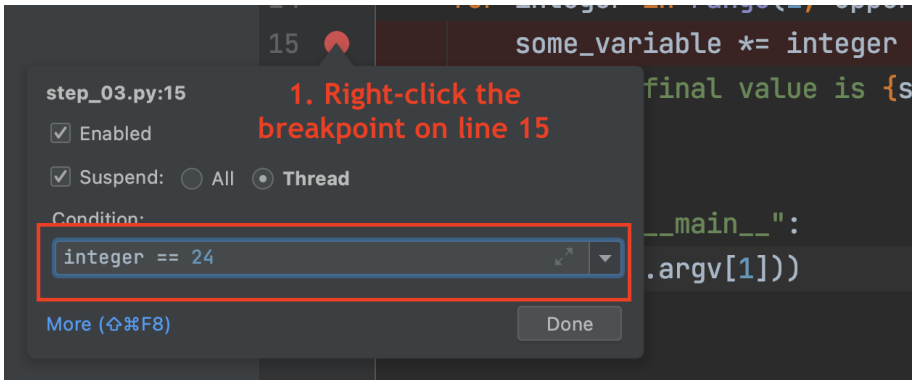

Figure 12: Add a breakpoint condition after you right-click a breakpoint.

(f) Did you notice the little question mark that PyCharm places on the breakpoint? This reminds us that it is a conditional breakpoint. Mouse over it (gently) to see the condition.

(g) **Debug the module again, and take a screenshot** when the debugger suspends execution of the thread. Make sure I can see these in the screenshot:

i. all the neat values appended to the lines of code in the editor pane, including the fact that integer is now equal to 24 or whichever value you selected

ii. the values stored in the variables in the main function's scope (visible using the Debugger sub-pane in the Debug pane) including integer.

(h) **Call the file step_03.pdf** and add it to git, commit it, and push it to the cloud.

(i) Neat! You can set a conditional breakpoint now. And you are starting to learn how to step through code. These are fundamental debugging skills.

4. **Now let's examine the module called step_04.py**:

(a) This module contains a recursive implementation of a Fibonacci number generator. It's recursive and thus superbly elegant, but there's lots of backtracking, so it's also superbly slow. I tried calculating Fibonacci(100) and immediately realized it was pointless. In fact, fibonacci(32) or so is about the boundary for what I consider to be fast enough.

(b) I want you to see how the function call stack looks when we invoke a recursive function. **Change line 21 to invoke fibonacci(30), and set a conditional breakpoint** on line 14 so that the debugger suspends execution of the thread when nth_term equals 1.

(c) **Take a screenshot of the call stack** in the Debugger sub-pane of the Debug pane. I want to see the stack of function calls.

(d) **Call the file step_04.pdf** and add it to git, commit it, and push it to the cloud.

(e) Watch the stack of function calls as you press Step Out again and again and again. Add a paragraph to your README.md. In one short ELI5 (Explain it like I'm 5 years old) paragraph, tell me what is happening to the stack of function calls. Why is it taking so long to shrink and return to main? Why is it growing and shrinking? Put this ELI5 paragraph under a header called *** Step 4 ***

5. **Let's finish with the module called step_05.py**:

(a) This module contains a sloppy function. The identifiers are not helpful. It is not documented. I have to carefully examine the code before I can guess what it does. Examine the function called do_something. What does it do?

(b) Setting a watch is exactly what it sounds like. It lets us watch a variable. The helpful thing is we can keep the variable in view while we examine other things in the list of variables local to each function in the call stack.

(c) **Debug the module**. Don't set a breakpoint. What we want to do instead is reveal the Debug pane. It should be empty, because we didn't set a breakpoint and the debugger executed the module to completion:

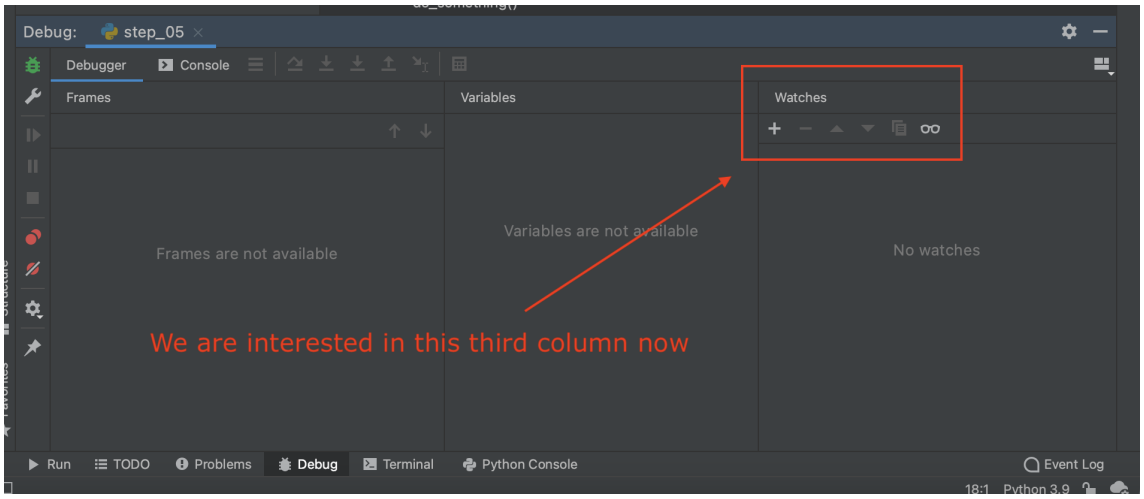

Figure 13: We debugged the module without setting a breakpoint to reveal the Debug pane:

(d) We can add a variable from the module to the Watches pane. It will remain there as we view other variables that belong to the functions in the call stack.

(e) **Click the + in the Watches pane** and add the variable result to the Watches list (press enter):

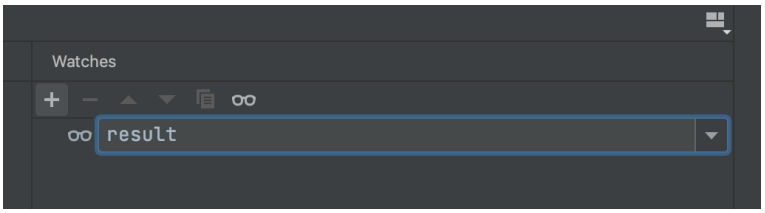

Figure 14: Let's watch the result variable as we loop through the function:

(f) **Set a breakpoint** on line 11 (the while loop) and begin debugging. Each time you reach the breakpoint, press Step Out to hurry through the loop. Keep an eye on the watch list.

(g) What happened? Did the function behave as you expected? Do you think there's a bug. I bet there's a bug. I bet you just revealed a bug. **Add appropriate documentation to the function that describes what you think it *should* do, and fix the bug**. Run it again to prove it works, then commit and push your change.

6. You now know about the following elements of debugging:

   (a) breakpoints and how to set them

   (b) conditional breakpoints

   (c) stepping through code

   (d) viewing the call stack and variable values

   (e) watching one (or more) variables closely during debugging.

7. There's more to debugging, but this is a very good start. Make ample use of breakpoints. Setting breakpoints, stepping over and into functions, and inspecting the program state, i.e., values stored in variables, is what developers do to study code that is probably not working.

8. Another trick: if a function yields an error, and it calls three helper functions, comment the helper functions out, then uncomment them one by one to help isolate the issue.

That's it! Good luck, and see you soon!