COMP 120 - Lab Session 05

Overview

In this lab session you will learn how to use a debugger in VS Code to help you trace through and fix flaws in your Python programs.

For this lab, you should work with your PSA2 partner.

If you run into any issues or have any questions during the lab, you may talk with your classmates or get the instructor's attention.

Initial Setup

To get started, we are going to clone a Git repository from GitHub. Open VS Code and in the command palette, enter "git clone" and select the option that comes up. When it asks you for the repository URL, enter the following:

```
https://github.com/usd-cs/comp120-sp24-lab05.git
```

It will then ask you for where to save the repository (store it some place logical, like a comp120 folder). VS Code will also ask you if you want to open the cloned repository. Select "Open" so that it opens in the current VS Code window.

If everything worked well, you should see the new repository folder opened in VS Code. If you get an error, talk with the instructor to help resolve the problem (a neighbor might be able to help also).

Debugging with Visual Studio Code

It is very rare that you will write a program and get it 100% correct from the very beginning. Instead, you should plan on having to debug your program to figure out why it isn't working as you expected.

While there are various ways to debug–including adding print statements at strategic places in your code to see if the values are what you expect–the most efficient way is to use a *debugger* to help you trace through your program and identify potential problems. Luckily, Visual Studio Code make it fairly easy to use a debugger with Python.

Start by opening up the search.py file in VS Code.

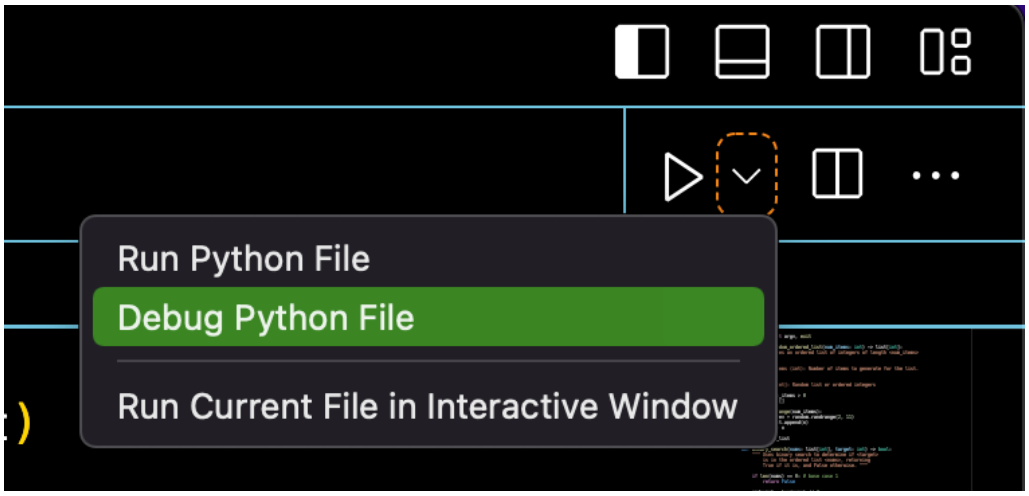
Run this Python program by clicking the "Play" button in VS Code. Note that this is a text-based program so check the terminal for any printouts and prompts.

Q1: What does this program do?

You can stop the program running by typing "CTRL-C" while in the VS Code terminal or "trashing" that terminal in VS Code. Do one of those now.

Q2: There are two functions located in this file. What are their names and what do they do?

Now we are going to switch to "Debug Mode" for running the application. To do that, select the drop down next to the "Play" button and select the "Debug python file" option, as shown below.



In the terminal, interact with the program as you normally would.

Q3: Does there appear to be any difference between how the program operates in the normal and debug run modes?

Stop the program from running now.

Setting Breakpoints

Now go to the end of the file and locate the "application code" contained in the `if __name__` conditional. The first non-comment line after this conditional is a call to the `create_ordered_random_list` function. If you put your cursor over the line number for this line, you will notice that a faint red circle will appear next to the line number. **Click that red circle**, which should make it both brighter and permanently on.

What you have just done is set a **breakpoint** in your program. A breakpoint is a line in your program where the debugger will automatically pause execution.

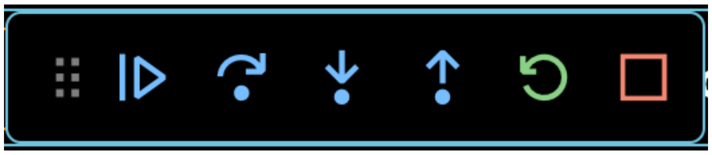
Click the Run button now, which should run the file in debug mode again.

Q4: Your program should be stopped at the line set by the breakpoint. What visual indicator is given to show that the program is stopped at the specified line?

IMPORTANT: It is critical to note that the program has stopped **before** executing the line it is currently on. For example, if this line contains an assignment statement, the assignment hasn't been completed yet.

Controlling The Program

Now that the program is stopped, we have control of what we want to do next. At the top of the screen, you should notice some "debug controls" that look like the following.



The options available to us at this point are:

1. Continue the program running. This is the first button, that looks like "Play".
2. Execute the current line of code and move onto the next. This is the second button with the curved arrow.
3. Execute the current line, "stepping into" the function being called if there is one. This is the third button with a down arrow.
4. Finish executing the current function. This is the fourth button with the up arrow.
5. Restart the program. This is the fifth button with the "loop back" icon.
6. Exit the debugger. This is the final button which is a square.

All of these are fairly intuitive, except the distinction between the second and third options, the "step over" and "step into".

Let's start by exploring the "step over" action (i.e. the curved arrow with the dot below it). Keep hitting that button until you get to the first line inside of the `while` loop.

Q5: What does this line of code do?

Click the "step over" button again, then make sure you enter a number in the terminal below (any integer will do).

Continue using the "step over" until you get back to the top of the `while` loop.

Now let's see what happens when we do the "step into" action. Let's restart the debugger by clicking the "Restart" debug button (i.e. the loop back circle icon). You should once again find yourself back at your initial breakpoint.

Now hit the "step into" button once.

Q6: What line of code did you jump to?

Keep using the "step into" button until you reach the `for` loop inside of `create_random_ordered_list`.

You have now seen the difference between the "step over" vs "step into" actions.

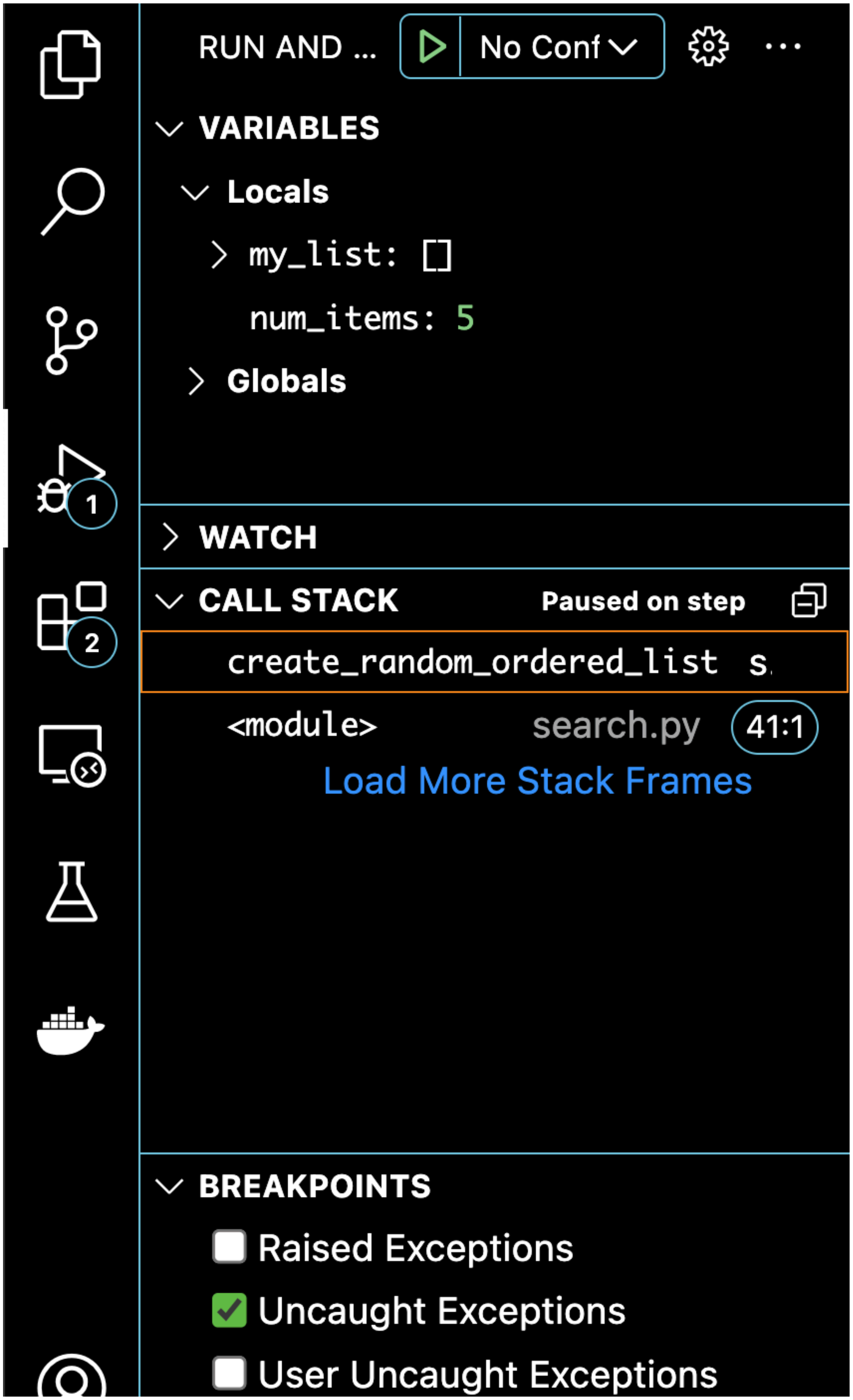
Q7: Summarize the difference between step over and step into, using your own words.

Share your answer in our CampusWire #labs chatroom, including the question number.

You may now be asking yourself, "Should I use step into or step over?" Because we use the debugger to help us fix problems, we want to focus our efforts on areas of the code where the bug is mostly likely coming from. If you have a function that has been well tested and shown to work correctly, you probably want to step over that function when debugging. If you have a function you are unsure about, it's probably best to step into that function so you can see if you can find the problem in there.

Variables and Call Stack

One of the other major benefits of using the debugger is the ability to easily track the contents of variables as well as the *call stack*. On the left side of your VS Code, you should see the "Debug Pane," which should look similar to the following.



RUN AND ... No Conf

VARIABLES

Locals

```
my_list: []
num_items: 5
```

Globals

WATCH

CALL STACK Paused on step

```
create_random_ordered_list  s
<module>                    search.py  41:1
```

Load More Stack Frames

BREAKPOINTS

- [ ] Raised Exceptions
- [x] Uncaught Exceptions
- [ ] User Uncaught Exceptions

At the top, you should see a section labeled "Variables," where you can see the contents of the current local variables/parameters and any global variables.

Below that (and a section labeled "Watch" that we'll ignore for now), you should see a section labeled "Call Stack". The first entry in this stack will be the current function.

Q8: How many entries are listed in the call stack right now?

Click on the second entry now.

Q9: Where in the code did it take you to? Why?

Share your answer in our CampusWire #labs chatroom, including the question number.

Now return to the most recent function by clicking on its entry in the call stack.

Q10: What variables are listed under "Locals" and what are their values?

Use the step over button to go through **two** iterations of the for loop, noting how the variables change after each step.

Q11: What are the contents of the `my_list` variable at this point?

While we could continue stepping through line-by-line until the function is done, let's use the "Step out" button. Click that button now.

Q12: What part of the program are you at now, and why?

Q13: What entries are there in the call stack now and why?

Tracing with the Debugger

We're now going to get a better sense for how the `binary_search` function works by tracing through it with the use of the debugger.

We'll start by creating a new breakpoint *inside* of the `binary_search` function, namely on the "`midpoint = (first+last)//2`" line. Let's now resume the program running at full speed. To do this, **click the "continue" debug** button (the debug icon that looks like "Play").

You might have expected it to have stopped at the breakpoint you just created, but instead you'll notice that the "Call Stack" lists the program as "running". That's because it's waiting for you to enter an integer in the terminal as part of the normal operation of this application. Review the list of numbers that was printed out and enter an integer that is **larger than** all of them.

You should now be at the newest breakpoint you set, which will be inside of `binary_search` function.

Q14: List the local variables and their values.

Let’s step through the first iteration of this loop by using the “step over” option until you return to the first line of the `while` loop. Pay attention to the values of the local variables as you do each step.

Q15: Which variables changed during this iteration and why?

Share your answer in our CampusWire #labs chatroom, including the question number.

Before doing anything else, make a prediction.

Q16: What will `first`, `last`, and `midpoint` if you were to complete one more iteration?

Let’s continue execution once more using the “continue” button.

Q17: Where are you at in the program and why?

Share your answer in our CampusWire #labs chatroom, including the question number.

At this point, you should still be on the line inside of `binary_search` where you set the breakpoint. If you are not at that point in the program, get the instructor or lab assistant’s attention for help.

Q18: Did the values for `first`, `last`, and `midpoint` line up with your prediction from earlier? If not, explain what happened.

Let’s make another prediction.

Q19: How many more iterations will the loop go for?

Share your answer in our CampusWire #labs chatroom, including the question number.

You will test your prediction by clicking the “continue” button until you reach a point after the loop is done, but don’t do that right away. First, add a new breakpoint on the “return False” line after the loop so that it’s easier to see when we’re done with the loop. Now go ahead and use continue, counting how many more iterations happen before the loop ends.

Let’s now repeat this process, but for a different target. - Click the “restart” button, which should take you back to your very first break point. - Click continue to get to your next breakpoint (inside of the `while` loop in `binary_search`). Remember that you will first get stopped at the point where you need to enter a number: enter a number that is **less than** every number in the list.

You should now be back at the top of the `while` loop in `binarch_search`.

Q20: How many iterations will this loop run for? Explain why.

Now check whether you are correct or not, using continue to run through each iteration of the loop.

IMPORTANT: Before continuing, check-in with the instructor or lab assistant. They will review your work and give you credit for this part of the lab.

Expecting Exceptions

In many cases, the code we write will have scenarios when we expect it to raise an exception. It is important to test that it actually does raise an exception in those cases. Luckily, pytest makes it fairly easy to specify when an exception should occur and what type of exception should be expected.

Open the `exceptional.py` file and locate the `foo` function.

Q21: For what value of n would this code raise an exception? What type of exception would it be?

Now open the `test_exceptions.py` function and located the `test_foo` test case.

Q22: How many times does this tester function call the `foo` function? What are the parameters given for each call?

To test the non-exception case, we use an `assert` statement. Because an exception isn't "returned" by a function, we can't use `assert` fo test that an exception occurred.

Q23: What syntax was used to indicate that we should expect an exception to occur in our test case?

Share your answer in our CampusWire #labs chatroom, including the question number.

Custom Exceptions

Python has many built-in exceptions for many common programming scenarios. However, sometimes we would like to have a custom exception for scenarios that are unique to the code we are writing. This will allow us to have a specific "except" block that catches only that exceptional scenario. Luckily, creating a custom exception type in Python is *exceptionally* easy.

Below demonstrates how you can create a custom exception named `MissingWalrusError`.

```
class MissingWalruseError(Exception):
   pass
```

That's all there is to it! If you wanted to create one named `TooMuchHomeWorkError`, you would simply replace the "MissingWalrusError" with "TooMuchHomeWorkError".

Open the `exceptional.py` file once again and find the `get_max` function.

Q24: What does this function do?

This function relies on the given file having two columns in it: one for name and one for amount. What we'd like to do is to have a special exception raised whenever the given file contains too many columns. We'll imaginatively call this the "TooManyColumnsError".

Inside of the `exceptional.py` create this new type of exception, adding it **above** the `foo` function.

Next, modify the `get_max` function so that it checks how many columns there are after splitting up the line. If it is more than 2, it should raise the `TooManyColumnsError` exception.

Now let's test that it works correctly. Inside of the `test_exceptions.py` file, add a new test case to test the `get_max` function. It should test two scenarios: one where the function is called with a valid file (`meow.csv`) and another where the file has too many columns (`bad_meow.csv`), so you would expect `TooManyColumnsError` to be raised.

Once you are done, run the tester file and check that both of the test cases passed.

IMPORTANT: Before continuing, check-in with the instructor or lab assistant. They will review your work and give you credit for this part of the lab.

Submission

On Canvas, go to the Modules page and find the entry Lab <#lab_num> Submission. On there, upload the following:

1. The answers to your lab questions (exported to PDF format from Google Docs).
2. All of the python files you modified as part of the exercises. To access these files, right click on one of them and select "Reveal in Finder/Explorer"; this will open up the folder containing these files.

Only one person in your pair needs to submit these files. The other person will also submit on Canvas **BUT** their submission will be a "Text Entry" containing only a comment stating with whom they worked for the lab.

PSA Work Time / Q&A

For the rest of this lab period, you should work on the current PSA. You may only leave early if you demonstrate to the instructor that your PSAs are all completed.