CMSC 201 Fall 2018
Lab 02 – Basic Debugging

Assignment: Lab 02 – Basic Debugging
Due Date: **During discussion**, September 10th through September 13th
Value: 10 points (8 points during lab, 2 points for Pre Lab quiz)

In Lab 1, we logged onto GL and set up folders for 201 in your **`home`** directory. We also created a simple Python program, and turned it in using the **`submit`** command. We'll be using many of these skills in this lab as well.

Part 1A: Review – Prior Assignments

Before going to Lab 2, you should have completed and understood both Lab 1 and Homework 0. You should have already created directories for your 201 files, including the main 201 folder, directories called Labs and Homeworks, and directories inside those with appropriate names (e.g., hw0, lab1, etc.).

Part 1B: Review – Commands in GL

You should already know a number of commands that you can use in GL's command line (the "terminal") from the previous assignments. We will briefly cover those, and will introduce a few other useful commands as well. You are not expected to master all of these, or to do so immediately — we are simply reminding you that these tools exist.

Command	Purpose and Example Usage
`cd`	Change your current directory Usage: **`cd lab2`** to move to the **`lab2`** directory (Use **`cd ..`** to move up one directory; use **`cd`** alone to go back to your **`home`** directory)
`ls`	List the contents of the current directory
`mkdir`	Create a new directory Usage: **`mkdir lab2`** to make a new **`lab2`** directory if it doesn't already exist
`mv`	Rename a file (i.e., "move" the file to a new name) Usage: **`mv oldName.py newName.py`** will rename the file from oldName.py to newName.py *Can also be used to rename directories*
`pwd`	Print the full path of the "working" (current) directory
`submit`	Allows you to submit assignments over GL Usage: **`submit cs201 HW1 hw1_part1.py`** submits the file hw1_part1.py to the **`HW1`** assignment for the **`cs201`** class

All of the commands above you've seen before. Here are some new commands that you might not have used yet, but that can be very helpful.

Command	Purpose and Example Usage
`cp`	Create a copy of an existing file Usage: **`cp existingFile.py newFileName.py`** will create a copy of the existing file, and name the copy "newFileName.py"
`clear`	"Clears" your screen by shifting your previous commands and output upward; you can still scroll up using the mouse or scrollbar
`"TAB"`	Hitting the tab key will auto-complete based on the available file or directory names. For example, typing "**`emacs la`**" and hitting tab will autocomplete "**`la`**" to "**`lab2.py`**" if the file exists
`"up arrow"`	Hitting the up arrow will recall your previous command to the terminal. Hitting it again will pull up the command before that one; repeat as necessary. You can also use the down arrow to go "back" a command if you go too far in your command "history."

There are many more useful commands that you can use in GL, and we'll mention them as they come up during the semester. If you see your TA or instructor using a command or shortcut that you don't know and would like to be able to use, ask them to explain it to you!

Part 1C: Review – Shortcuts for emacs

Finally, let's cover some basic emacs shortcuts. Again, you will already know some of these from doing previous assignments, but some of them will also be new to you. You do not "need" to know any of them beyond the first two (how to save your file and how to exit emacs), but mastering a few more will make your programming experience more enjoyable.

Command	Meaning
`CTRL+X, CTRL+S`	Save the file and **stay** in emacs
`CTRL+X, CTRL+C`	Save the file and **close** emacs
`CTRL+_`	Undo your last edit; use it again to undo the previous one as well (Control + Shift + "-" to create an underscore)
`CTRL+K`	Cut everything on the line *after* the cursor ("kill")
`CTRL+Y`	Paste the text cut by the **`CTRL+K`** command ("yank")
`CTRL+A`	Go to the *beginning* of the current line
`CTRL+E`	Go to the *end* of the current line

Part 2: Exercise: Programming from Scratch

As we've discussed in class, testing and debugging your programs is a large part of being a successful programmer. Sometimes, you may even have to debug other people's code!

In this lab, we'll be creating two files: **`lab2.py`** will be a file you create, and **`errors.py`** will be a file you copy into your directory, before finding and fixing the errors it contains. Both files will be counted as part of the grade for Lab 2.

In this lab, you'll learn how to copy files into your account from an instructor's directory, and you'll write your first complete Python program.

Tasks

- ☐ Download the **`.emacs`** file
- ☐ Create a **`lab2.py`** file from scratch
- ☐ Fix any errors in the **`lab2.py`** file
- ☐ Download the **`errors.py`** file
- ☐ Fix all of the problems and bugs in the **`errors.py`** file
- ☐ Show your work to your TA

Part 3A: Downloading the .emacs File

The first thing you'll do is download a file that will configure the emacs editor we'll be using, to customize the way the emacs program behaves. While in your *home directory*, copy the `.emacs` file into your current folder by using the `cp` command below (`cp` simply stands for "copy").

```
cp /afs/umbc.edu/users/k/k/k38/pub/cs201/.emacs .
```

There are three parts to the command, and all three are important:

1. "`cp`" is the command, and in this case it stands for copy
2. "`/afs/umbc.edu/users/k/k/k38/pub/cs201/.emacs`" is where the file you are copying is located
3. "`.`" (a single period) is where the file will be copied to
   (The period means it will be copied to the current folder, and will keep the same filename.)

The period "`.`" in front of the file name indicates the file is a hidden file. If you simply type `ls`, you won't see it listed. To double check that you successfully copied the file, you need to use the command "`ls -a`". The `-a` means "all" and will show all the files in that directory, even hidden ones.

Part 3B: Creating a File from Scratch

Next, you are going to create your first complete Python file, entirely from scratch. First, create the **`lab02`** folder using the **`mkdir`** command -- the folder needs to be inside your **`Labs`** folder as well. *(For a reminder of how to create and navigate folders, refer to the instructions for Lab 1.)*

Next, create a file called lab2.py by using the "touch" command, and then open it for editing with emacs:

```
touch lab2.py
emacs lab2.py
```

You'll want to reproduce all of the text below inside your **`lab2.py`** file, making sure to include all of the "**#**" signs, and to follow the capitalization shown.

(This is an image, so you'll need to re-type it by hand.)

```
# File:         lab2.py
# Author:       YOUR NAME
# Date:         9/TODAY/2017
# Lab Section:  YOUR_DISCUSSION_SECTION
# UMBC email:   YOUR_EMAIL@umbc.edu
# Description:  This program shows the proper layout of code in a Python file,
#               and greets the user with the name of the programmer.

def main():

    # introduces the programmer
    print("Hello, my name is YOUR_NAME)

main()
```

Quick note about collaboration in labs: If you get stuck or have a question, you are welcome to work with the other students in your lab section. We do not require that you fill out the Collaboration Log for collaboration that occurs during a discussion. You should still not copy code, and you should not type on another student's keyboard. You should also **never email anyone code** (from a lab or from a homework assignment) for any reason.

Part 3C: About the Header Comment

The pound symbols "**#**" you have in the lab2.py file are used to tell Python that any text on that line after the pound sign is a ***comment***. Comments are ignored by Python, and the text following a pound sign does not need to follow any of Python's syntax rules. Comments are useful for the person reading the code (you, your TA, your instructor, etc.).

Programmers use comments to explain what the code is doing, to leave notes to themselves, and to document things about the code. For example, the comments at the top of the file are called a "header comment block," and record who created the file, when, and what the file is supposed to do.

We'll talk more about comments throughout the semester, since they are an incredibly important part of programming and being a good programmer.

Part 4: Exercise: Finding and Fixing Errors

In this part of the lab, you'll be working on a Python file full of errors. We'll first explain how to find them, and how to understand the error messages. Following that, you'll solve the errors on your own.

Part 5A: Running the lab2.py File

First, let's try running the lab2.py file you have created. Save your file and exit emacs. Go ahead and run your program by using the **`python`** command:

```
python3 lab2.py
```

If you copied everything exactly, your program won't run — instead, you'll get an error message.

Part 5B: Reading Error Messages

Your error message should look something like this:

```
linux3[11]% python3 lab2.py
  File "lab2.py", line 12
    print("Hello, my name is YOUR_NAME)
                                       ^
SyntaxError: EOL while scanning string literal
linux3[12]%
```

There are a couple key pieces of information in this error message:

1. We are told the name of the file, and the line where the error occurred In this case, it's our **`lab2.py`** file, and the error is on line 12
2. Python has attempted to pinpoint the error even further for us, using the "**^**" symbol you can see on the second-to-last line
3. Python has told us what kind of error it is, and some details:
    a. It is a syntax error
    b. Python reached EOL (End of Line)
    c. While scanning a "string literal"

Part 5C: Fixing an Error

Hopefully you've spotted the error already – the **`print()`** statement is missing the closing quotation marks. To fix it, we'll need to open **`lab2.py`** again for editing.

Once you do that, take a look at the bottom of the screen, and you should see something like this:

The "**L1**" you see there (second from the right) stands for "Line 1" – the error was on line 12, so move your cursor down until you've reached "**L12**" instead. Fix the error by adding the closing quotation mark, and save and exit again.

Try running your program again – if you fixed the error correctly, it should run without any errors, and display the greeting you wrote.

If it doesn't work, use what you've learned to find the "next" error and fix it. Python often will only display the first error it finds, so you may find yourself having to do this multiple times when working on your assignments as well.

(If there is more than one error, start with the message at the bottom first. Fix that one error, and then try to run your program again.)

PRO TIP:
Python won't be able to spot logical errors for you, but it tends to be very good at pinpointing syntax errors. However, it sometimes won't notice an error has occurred until a few lines after the actual error. If you don't see anything wrong with the line Python has indicated, try looking at the lines directly before it for anything odd.

Think about it like this: if you made a wrong turn while driving, you might not notice your mistake until you came to an unexpected dead end. But the dead end itself is not the error — it's one of the turns you took before that.

To see an example, **try deleting the closing parenthesis** at the end of the print statement on line 12. Python will now mark line 14 as the location of the error, because that is where it first realizes that something is wrong.

Part 5D: Applying Your Debugging Skills

Now it's time to put your bug fixing abilities to the test! Copy a file called **`errors.py`** into your **`lab2`** folder using the **`cp`** command. (Don't forget to include the period at the end of the command!)

```
cp /afs/umbc.edu/users/m/n/mneary1/pub/cs201/errors.py .
```

Before you jump into trying to fix the bugs, take a moment to read the code and figure out what the program should be doing. Then use your new knowledge of finding and fixing bugs to update the **`errors.py`** file to run without any errors. (You might have to try multiple times to run the file before all of the errors are found and fixed – keep trying!)

Part 5D: Testing Your Fixes

Test your fixed **`errors.py`** Python program with different inputs to ensure that it runs correctly.

If your testing finds a bug, fix it, and try running the program again.

(*HINT:* Try testing out the program with the numbers 5, 6, and 7 as input. What average does your program calculate? What is the correct answer?)

Once **`errors.py`** is fixed, add two lines to the file's header comment block.

```
# Fixed by:    YOUR_NAME (YOUR_EMAIL@umbc.edu)
# Date fixed: TODAYS_DATE
```

The headers of your files (the block of comments at the top) are very important. Double check that you have correctly completed the headers for the **`errors.py`** and **`lab2.py`** files.

Part 6: Completing Your Lab

Since this is an in-person lab, you do not need to use the **`submit`** command to complete your lab. Instead, raise your hand to let your TA know that you are finished.

They will come over and check your work – they may ask you to run your program for them, and they may also want to see your code. Once they've checked your work, they'll give you a score for the lab, and you are free to leave.

Tasks

As a reminder, here are the tasks again:

- ☐ Download the **`.emacs`** file
- ☐ Create a **`lab2.py`** file from scratch
- ☐ Fix any errors in the **`lab2.py`** file
- ☐ Download the **`errors.py`** file
- ☐ Fix all of the problems and bugs in the **`errors.py`** file
- ☐ Show your work to your TA

IMPORTANT: If you leave the lab without the TA checking your work, you will receive a **zero** for this week's lab. Make sure you have been given a grade before you leave!