CMSC 201 Fall 2017

Lab 09 – Advanced Debugging

Assignment: Lab 09 – Advanced Debugging
Due Date: **During discussion**, October 30th through November 2nd
Value: 10 points (8 points during lab, 2 points for Pre Lab quiz)

This week's lab will give you practice with debugging more complex problems, such as logic errors, that you may start encountering.

(Having concepts explained in a new and different way can often lead to a better understanding, so make sure to pay attention as your TA explains.)

Part 1A: Review – Introduction to Errors

Throughout this semester, we have been working to program a variety of different small applications and projects to practice using Python. When we make a mistake in our code, we have attributed it to one of two types of errors.

The first kind, ***syntax errors***, happen when Python can't understand the code as it is written, due to missing pieces, indentation errors, or typos. The second kind, ***logic errors***, happen when Python can run the code, but it doesn't behave as the programmer (you) expected or wanted it to behave. A third type of error, one that we haven't really discussed yet, is called a ***run-time error***. Run-time errors can be either syntax or logic errors.

With syntax-based run-time errors, they aren't found until the code has actually started running. (This is often because the syntax error is contained in code inside a conditional, or a function that may not always be called.) And with logic-based run-time errors, they can cause the program to run forever (infinite loop), incorrectly (conditionals that evaluate to False instead of True), or to exit unexpectedly due to other issues.

We are going to look at each kind of error and some techniques that we use to try and debug them.

Error Type	Amount of Code that Runs	What Catches the Error?	Difficulty to Debug
Syntax Error	No code will run	Python Interpreter	Usually Easy
Logic Error	All code will run	User	Usually Difficult
Run-Time Error	Some code will run	Python Interpreter User	Both Easy and Difficult

Part 1B: Review – Syntax Errors

Syntax errors are the most common type of error because there are a variety of small mistakes may cause them. In plain English, we can think of syntax errors as "sentences" that do not make grammatical sense. They are missing parts of the sentence, whether that is a noun (*i.e.*, a variable name), a verb (*i.e.*, a function call), or a punctuation mark (*i.e.*, a quotation mark or parentheses). They might also have extra pieces that confuse the sentence's meaning, or have pieces swapped entirely.

In English, a syntax error might look like this:

Dog cat bear pizza

This sentence doesn't make any sense – it's simply a list of nouns with no verbs or prepositions, and it doesn't even have a period at the end. Syntax in programming is similar to grammar in English.

Common syntax errors in Python include:

1. Mismatched parentheses
2. Incorrect capitalization
3. Indentation issues
4. Missing colons
5. Missing or incorrect quotes

Part 1C: Review – Logic Errors

Logic errors are a type of error where the code will execute but the results of the code are not what you expected. Python doesn't "care" about logic errors, and so the error is only discovered when the programmer examines the output of the program, and finds it different from what they expected.

A common example of a logic error in programming is when a programmer uses a "**=**" instead of a "**==**". We know from our practice that "**=**" is the assignment operator and sets the value of one variable to the value in the expression following. We also know that "**==**" is a comparison operator that returns either **`True`** or **`False`** after comparing two expressions or variables.

Python actually will catch this error for you, but this is partly because it is such a common error, and Python itself has been programmed to look for it. Other similar errors, such as mixing up "**<**" and "**>**" or using "**==**" instead of "**!=**" will not be caught by Python.

In English, we might think of these like this:

Let's eat grandma!

vs.

Let's eat, grandma!

Both statements make grammatical sense, but the additional comma changes the meaning of the statement. Missing the comma in the sentence would be similar to a logic error in Python.

Common logic errors in Python include:

1. Order of operations (PEMDAS)
2. Reversing comparison operators
3. Using the wrong Boolean operators (**`and`** and **`or`**)
4. Integer division vs "regular" division
5. Mixing up modulus and integer division
6. Infinite loops

Part 1D: Run-time Errors

Run-time errors happen when Python understands what you are saying, but runs into trouble when following your instructions. Importantly, the code will run for some amount of time before failing. All run-time errors are caused by either a syntax error or logic error at their core.

In English, we might think of a run-time error as:

Put the hippo in the fridge.

From a grammatical standpoint, we understand what the sentence is asking us to do, but when we go to actually do it, we would fail. Similarly, when programming, Python understands what the code is supposed to do, but is unable to actually execute what is written when the time comes.

For example, we can imagine a scenario where we write a program to calculate an average given a list of grades. If the list of grades is empty, and we try to divide by the length of the list, we are now asking Python to divide by zero, which isn't possible. The code runs, but at some point it is asked to do something that it doesn't know how to do, and so it fails.

Common run-time errors in Python include:
1. Using an undefined variable or function
2. Dividing by zero
3. Using operators on the wrong datatype (*e.g.*, `"4" + "2"`)
4. Referencing a variable before assignment
5. Trying to combine two variables of incompatible types without conversion

Part 1E: Review – Debugging in Python

Being able to discern what type of error you are dealing with will give you an idea of what you may need to debug the program.

Generally, **syntax errors** will be caught by the Python interpreter, and the interpreter will give you some hints on where to look to fix the errors.

Because **logic errors** are syntactically correct, the Python interpreter does not actually "see" the error, and won't give many hints on how to fix it.

Run-time errors are sometimes identified by the Python interpreter. However, actually fixing the problem is up to you – you'll need to figure out when and where the error is occurring in the code, as well as how it should be fixed.

Part 1F: New Material – Debug Statements

A "debug statement" is a **`print()`** statement that gives you more information on what exactly is going on. For example, you might want to see:

What number a **`while`** loop is accessing each time it loops over a list:

```
print("At index", index, "the number is", nums[index])
```

The types of two variables that are being compared:

```
print("Comparing", type(var1), "to", type(var2) )
```

Whether a conditional is being entered as expected:

```
if chosenAction == "dance":
    print("Entered the 'dance' conditional")
```

If you place a meaningful **`print()`** statement inside your code, it can show you what is going on in the "background" of your program. Each time the code is run, the information in your debug statement will be printed to the screen, allowing you to trace what is happening with your program.

Debug statements can be anything that helps you figure out what your program is actually doing. Just don't forget to remove them when you're done!

Part 2: Exercise

In this lab, you'll be downloading a program that has a large number of errors that need to be fixed. The original programmer didn't use incremental development, and definitely didn't test as they completed each function or piece – so it's up to you now!

Tasks

Starting:

- ☐ Copy the **`broken_lab09.py`** file from Dr. Gibson's **`pub`** directory
  - ☐ It should have been renamed to be **`fixed_lab09.py`**

Debugging:

- ☐ Open the file and examine the functions and **`main()`**
- ☐ Fix any syntax errors you see
- ☐ Fix any logic errors you see
- ☐ Run the program
  - ☐ Read and understand the error message(s)
  - ☐ Fix the (syntax) error reported by Python **or** use debug statements to pinpoint where and why the (logic) error is occurring

General:

- ☐ Repeat debugging steps until program runs with no errors
- ☐ Show your work to your TA

Part 3A: Downloading the File

First, create the **`lab09`** folder using the **`mkdir`** command – the folder needs to be inside your **`Labs`** folder as well.

Next, copy a file into your **`lab09`** folder using the **`cp`** command. (The command should be all on one line.)

```
cp /afs/umbc.edu/users/k/k/k38/pub/cs201/broken_lab09.py
fixed_lab09.py
```

This will copy the file **`broken_lab09.py`** from Dr. Gibson's public folder into your current folder, and will change the file's name to **`fixed_lab09.py`** instead.

Part 3B: Debugging the Program

The first thing to do when debugging a program (yours or someone else's) is to make sure you understand what the programmer intends for the code to do. Open the file and read the code and the comments.

You will probably spot some ***syntax errors*** just by looking at the code – go ahead and fix any you see. You may also spot some ***logic errors*** in the functions as well, especially with regards to input and output for the different functions.

Once you've fixed all of the errors you can see, go ahead and run the lab. You'll probably get something like this:

```
linux1[5]% scl enable python33 bash
bash-4.1$ python lab09.py
  File "lab09.py", line 50
    def numsEquiv(item1, item2)):
                                ^
SyntaxError: invalid syntax
```

Python is telling us that there is a **syntax error** on **or above line 50**. Specifically, it believes that the error is due to the second parentheses at the end of the function definition. Fix this error, and then exit and try again to run the program.

Here are some shortcuts that may make your life a bit easier while debugging. The first two shortcuts work best after Python 3 has already been enabled, and prevent you from having to close and re-open the file every time you want to run the program. (You still need to save before minimizing.)

Command	Meaning
CTRL+Z	"Minimize" the file you're working on in emacs
fg	From the command line, "maximize" the previously minimized file
META+G+G+number	Go to the line number specified (after hitting enter) For example, **META+G+G+19** moves to line 19

After fixing that and a few other errors, you might see a message like this:

```
bash-4.1$ python lab09.py
Traceback (most recent call last):
  File "lab09.py", line 96, in <module>
    main()
  File "lab09.py", line 77, in main
    num1 = getValidInt()
  File "lab09.py", line 25, in getValidInt
    num = input(int(msg))
ValueError: invalid literal for int() with base
10: 'Enter an integer between 1 and 100
(inclusive): '
```

This error message looks a lot more complicated than the last one, but it's actually not! In an attempt to help, Python is "tracing back" the whole journey of how it saw the error. If we read from the top, we can see that the following things happened (in this order):

1. Python called **`main()`**, found on **line 96**
2. From inside **`main()`**, Python called **`getValidInt()`** on **line 77**
3. Inside **`getValidInt()`**, Python found a value error on **line 25**

When we look at the message for the value error (a type of run-time error) we see that the **`int()`** function is given an "invalid literal" – specifically, that is expecting something of "**base 10**" (a decimal) but is instead receiving the string "**Enter an integer between 1 and 100 (inclusive):** ". Fix this error, and then exit and try again to run the program.

Continue with this cycle until everything works correctly:

1. Try to run the program.
2. Read the error message.
    a. Find the problem in your code.
    b. Fix the problem.
    c. Go back to step 1.
3. If there is no error message, test that the code functions properly.
4. If it doesn't function properly, go back to the code and examine its behavior. Debugging/print statements can be helpful.

Once **`getValidInt()`** functions correctly, go down to **`main()`** and uncomment the three lines of code after the "**`# check for duplicates...`**" comment. Start the cycle again: run, fix the errors Python catches, and test to make sure there aren't any remaining errors.

Once **`twoInARow()`** functions correctly, go back down to **`main()`** and uncomment the two lines of code after the "**`# check to see if...`**" comment. Start the cycle again: run, fix the errors Python catches, and test to make sure there aren't any remaining errors.

And finally once **`equiv()`** functions correctly, uncomment the last two lines in **`main()`**, and test that **`average()`** works correctly. Use the same debugging techniques you used for the earlier parts of the lab.

Part 4: Completing Your Lab

To test your program, make sure that you've enabled Python 3, then run **`lab09.py`**. Try a few different inputs to see how well your program works. The two example runs below should test that everything works.

```
bash-4.1$ python lab09.py
Enter an integer between 1 and 100 (inclusive): 200
Invalid choice!
Enter an integer between 1 and 100 (inclusive): -5
Invalid choice!
Enter an integer between 1 and 100 (inclusive): 1
Thank you for choosing 1
Found dupes of 4 next to each other.
Found dupes of 7 next to each other.
The result of the nearby duplicate test:
There are 2 matches
The result of the equivalence test: No match
The average is 4.090909090909091

bash-4.1$ python fixed_lab09.py
Enter an integer between 1 and 100 (inclusive): 9
Thank you for choosing 9
Found dupes of 4 next to each other.
Found dupes of 7 next to each other.
The result of the nearby duplicate test:
There are 2 matches
The result of the equivalence test: They match!
The average is 4.090909090909091
```

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 list can be found on next page.)

Tasks

Starting:

- ~~☐ Copy the **broken_lab09.py** file from Dr. Gibson's **pub** directory~~
  - ~~☐ It should have been renamed to be **fixed_lab09.py**~~
- ☐ Copy the **less_broken.py** file from Dr. Gibson's **pub** directory, *and savor the function headers, spaces between operators, and useful variable names and comments used in the program's code*
  - ☐ It should have been renamed to be **lab09.py**

Debugging:

- ☐ Open the file and examine the functions and **main()**
- ☐ Fix any syntax errors you see
- ☐ Fix any logic errors you see
- ☐ Run the program
  - ☐ Read and understand the error message(s)
  - ☐ Fix the (syntax) error reported by Python **or** use debug statements to pinpoint where and why the (logic) error is occurring

General:

- ☐ Repeat debugging steps until program runs with no errors
- ☐ 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!