UMBC
AN HONORS UNIVERSITY IN MARYLAND

CMSC 201 Spring 2016
Lab 09 – Debugging

Assignment: Lab 09 – Debugging
Due Date: During discussion, April 11th through April 14th
Value: 10 points

Part 1: 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 what you are saying. The second kind, ***logic errors***, happen when Python can run your code without crashing but still produces the wrong results. A third kind, one that we haven't really discussed yet, is called a ***run-time error***. We are going to look at each kind of error and some techniques that we use to try and debug them.

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

Part 2: 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 sense. They are missing parts of the sentence such as the noun and the verb or perhaps a punctuation mark.

In English, a syntax error might look like this:

Dog cat bear pizza

From a grammatical standpoint, this doesn't make much sense and it is missing a period. It is even missing a verb. Syntax means the same as grammar.

Common syntax errors in Python include:

1. Unbalanced parentheses
2. Capitalization issues
3. Indentation issues
4. Missing colons
5. Missing or incorrect quotes

Part 3: 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. We see this type of error when the code appears to be doing what you expect but when you check the output, it differs from the expected output. 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 know from our practice that "==" is a comparison operator and it returns either true or false after comparing those two values. We can imagine a scenario where our code will run with the wrong operator causing significant errors in our code.

In English, we might think of these like this:

Let's eat grandma!
vs.
Let's eat, grandma!

Both statements make sense, however, the additional comma changes the meaning of the statement. This would be similar to a logic error in Python.

Common logic errors in Python include:

1. Order of Operations (PEMDAS)
2. Misusing assignment operator (=) and comparison operator (==)
3. Integer Division
4. Infinite loops

Part 4: 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.

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

Put the hippo in the fridge.

From a grammatical standpoint, we understand what we are trying to do, but when we go to actually do it, we would fail. Similarly, when programming, we can imagine a scenario where we are asked to calculate a grade for someone and we are inputting scores for them and when we are asked for a score, we don't enter anything and later in the code (during the averaging section) we try to divide by zero. The code ran but at some point it was asked to do something that it didn't know how to do.

Common run-time errors in Python include:

1. Referencing an undefined variable or function.
2. Dividing by 0.
3. Using operators on the wrong datatype.
4. Referencing a variable before assignment, especially in the context of functions
5. Trying to combine two variables of incompatible types without conversion, e.g. adding strings and integers.

Part 5: Debugging in Python

Depending on what type of errors you are dealing with really gives you some insight into how we may need to debug the program. Generally, syntax errors will be caught by the Python interpreter and the interpreter will give you some hints to where to look to fix the errors. Because logic errors are not obvious and are syntactically correct, the Python interpreter is not going to give any hints as to how to fix the errors. Run-time errors are first identified by the Python interpreter, however, the debugging is often left to finding how why that situation occurred and therefore is reliant on the user.

Type of Error	What Catches Error?	Common Tools to Debug
Syntax Error	Python Interpreter	Python Interpreter
Run-Time Error	Python Interpreter User	Python Interpreter, pdb, User, IDE
Logic Error	User	pdb, User, IDE

So far this semester we have only used our Python interpreter to debug our code. As you can imagine in complex code, debugging complex code is difficult especially when we are trying to fix a logic error that only occurs in very specific (or uncommon) scenarios. The good news is that Python has an available debugger (that is built into GL) called Python DeBugger (pdb). We will spend a few minutes trying to figure out some of the simple ways to use pdb.

Here is a step-by-step example:

1. Start with a simple example sum.py

```
a = 1234
b = 2345
c = 3456
final = a + b + c
print final
```

2. In order to enable the pdb debugging program, we need to insert a command called import pdb at the very beginning of our code.
3. Now, we can place our trace command to show where we want to start examining our code, line-by-line.
4. The new code in sum.py looks like this:

```
import pdb
a = 1234
pdb.set_trace()
b = 2345
c = 3456
final = a + b + c
print final
```

5. Now save and exit emacs and try and run your code:

```
-bash-4.1$ python sum.py
> /afs/umbc.edu/users/j/d/jdixon/home/CMSC201/lab9
/sum.py(4)<module>()
-> b = 2345
(Pdb)
```

6. Notice how the code stops executing on the place we set the pdb.set_trace(). This allows us to check the values of the variables at a specific location in the code.
7. The (Pdb) tells us that we are in a special debugging command line where we can use some specific commands to help us debug our code.
    a. Common Commands in PDB
        i. list – shows the code and where we are in the execution
        ii. w – shows where
        iii. n – next and goes to the next line in the code
        iv. q – quits the pdb debugger
        v. c – continues to the end of the code
        vi. p <variable name> – (print intCounter) prints a variable (a variable only appears after that line is executed!)
8. Let's test some of these commands:

```
-bash-4.1$ python sum.py
> /afs/umbc.edu/users/j/d/jdixon/home/CMSC201/lab9/sum.py(4)<module>()
-> b = 2345
(Pdb) n
> /afs/umbc.edu/users/j/d/jdixon/home/CMSC201/lab9/sum.py(5)<module>()
-> c = 3456
(Pdb) n
> /afs/umbc.edu/users/j/d/jdixon/home/CMSC201/lab9/sum.py(6)<module>()
-> final = a + b + c
(Pdb) n
> /afs/umbc.edu/users/j/d/jdixon/home/CMSC201/lab9/sum.py(7)<module>()
-> print (final)
(Pdb) p final
7035
(Pdb) list
  2     a = 1234
  3     pdb.set_trace()
  4     b = 2345
  5     c = 3456
  6     final = a + b + c
  7  -> print (final)
[EOF]
(Pdb)
```

Part 6: Practice Debugging Some Code

For the hands-on part of the lab, we are going to use our techniques to try and debug three snippets of code**. You may use pdb if you want to help the debugging process, but you are not required to use for this lab.**

Work on the first file, then after about 10 minutes whether you figured it out or not your TA will show you how to debug the code! This may seem easy but debugging is one of the most important skills you can learn, so pay attention!

A. Copy the code below into a file called: **`letterCount.py`**
   All errors are in the letterCount function.

```
#letterCount should take a letter and word,
#and count the number of times letter appears
#in the word
def letterCount(myString):
    for 1 in myString:
        if i = letter:
            count = 1
    print(count)

#no errors below this line
def main():
    print("Should print: 3\nPrints: ",end="")
    letterCount("a","aardvark")
main()
```

B. Copy the code below into a file called: **gcd.py**
There are all sorts of errors in this snippet. Can you find/fix them all?

```
# Python program greatest common divisor (GCD) of two numbers.
# The GCD is the largest positive integer that perfectly
# divides the two given numbers.

def gcd(x, y):
   if x > y:
       smaller = y
   else:
       smaller = x
   gcd = 0
   for i in range(1,smaller):
       if((x % i == 1) and (y % i == 1)):
           gcd = i
   return gcd

#No errors below this line
def main():
   print("Should print: 15\nPrints: ",end="")
   print(gcd(30,15))
   print("Should print: 7\nPrints: ",end="")
   print(gcd(7, 49))
   print("Should print: 45\nPrints: ",end="")
   print(gcd(45,45))
main()
```

C. Copy the code below into a file called: **palindrome.py**
This has a big logic error so make sure you understand the code!

```
#tests whether myString is a palindrome
def isPalindrome(myString):
    tempString = myString
    length = len(tempString)
    half = length/2
    isPalindrome=True
    for i in range(int(half):
        if myString[i] != myString[length-i]:
            isPalindrome = False
    print(isPalindrome)

#no errors below this line
def main():
    print("Should print: True\nPrints: ",end="")
    isPalindrome("oozyratinasanitaryzoo")
    print("Should print: False\nPrints:", end="")
    isPalindrome("18101181")

main()
```

Part 7: Completing Your Lab

To test your program, first enable Python 3, then run **letterCount.py**. The code should run. Then run **gcd.py.** The code should run. Finally, run **palindrome.py.** The code should run.

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. **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

pdb Commands

Startup and Help

outside a python file:

`python -m pdb <name>.py[args]`	begin the debugger
`help [command]`	view a list of commands, or view help for a specific command

within a python file:

```
import pdb
...
pdb.set_trace()
```

begin the debugger at this line when the file is run normally

Navigating Code (within the Pdb interpreter)

`l(ist)`	list 11 lines surrounding the current line
`w(here)`	display the file and line number of the current line
`n(ext)`	execute the current line
`s(tep)`	step into functions called at the current line
`r(eturn)`	execute until the current function's return is encountered

Controlling Execution

`b [#]`	create a breakpoint at line [#]
`b`	list breakpoints and their indices
`c(ontinue)`	execute until a breakpoint is encountered
`clear[#]`	clear breakpoint of index [#]

Changing Variables / Interacting with Code

`p <name>`	print value of the variable <name>
`!<expr>`	execute the expression <expr> NOTE: this acts just like a python interpreter
`run [args]`	restart the debugger with sys.argv arguments [args]
`q(uit)`	exit the debugger

From: http://web.stanford.edu/class/physics91si/2013/handouts/Pdb_Commands.pdf