

CSCI 141
Spring 2021

Lab 2: Type Conversions, Operators and Operands, Print Formatting,
Errors

Due Date: Friday, April 16th at 10:00pm

Overview

This lab introduces you to the Linux command line, command line arguments, and explores errors, printing and operands. You should have enough time to complete this lab during the lab session. Upload your submission to Canvas before the deadline. If you have questions, be sure to ask the TA. Ask questions often. Labs are your opportunity to get personalized help!

While we are learning remotely and not able to access the CS Labs on campus in person, the Linux component of this lab is **optional**, but recommended. If you're considering going on in CS, you'll need these skills later, so I highly recommend following these instructions to complete the steps using `repl.it`. Unfortunately, it requires creating a (free) account, so I'm not going to require it. If you want to skip this portion, you will still need to read Section 3, and try out the code example in Thonny, running it as described in Step 7.

1 Getting a Linux-like environment using `repl.it`

`repl.it` is a web-based service that provides an online IDE, similar to Thonny, that allows you to write and test programs right in the browser. It also allows you to interact with a linux command line, which is why we'll be using it for this portion of the lab instead of Thonny.

Head over to <http://repl.it> and sign up for a free account. Log in and press the "+ New REPL" button. Select Python as the language, then "Create repl" to create a Python programming environment.

You should land on a screen with three columns - one is a file browser (showing `main.py`, one is an editor where you can write Python programs (it currently has `main.py` open), and the right-most column has two tabs, "Console" and "Shell". The Console tab is similar to Thonny's Shell pane - this is a Python REPL (read-eval-print loop) where you can run snippets of Python interactively. Select the Shell tab; this takes us to a command-line shell (also known as a *terminal* similar to one you'd find on the Linux operating system.

2 Linux Command Line Basics

Windows, Mac OS, and Linux all provide graphical interfaces such as those you're used to using, that allow you to open programs and accomplish tasks using the mouse, icons, windows, and so on. All of these operating systems also provide another way of interacting with them, called a *Command Line Interface* (CLI). Although there is a steeper learning curve for knowing how

to accomplish things using the command line, for certain tasks it can be more powerful once you know how to use it.

In this lab, you will learn the very basic elements of how to interact with the Linux command line and learn how to run Python code without the help of an IDE such as Thonny. What you will learn here is only a tiny fraction of the commands available; you can find a helpful "cheat sheet" of many commonly used Linux commands on the course website¹ if you want to learn more.

1. In the terminal, you'll see a **prompt** that begins with `/`, then shows the name of the environment you're in, and ends with a `$` sign. This is called the command line, or command prompt; you'll type commands here to interact with the system. Commands that you issue are interpreted by a program called a shell (the default shell, or command line language, in the lab machines and also repl.it is bash). It is one of the many shells available in Linux.
2. All of the things that you can do with the mouse when interacting with a windows environment you can also accomplish via the command line. In the following steps you will create a new directory (folder), navigate into that folder, and run Python from the command line. For these instructions, code and/or sample file content or output are shown in boxes. Type commands EXACTLY as they provided, and press return/enter to issue the command. For example:

```
whoami
```

is instructing you to type the command `whoami` on the command line and execute it by pressing return/enter. Try it out. This command shows the username of whoever's logged into the shell. In repl.it, it shows the username `runner`; on a CS lab machine, it would show you your Western username instead.

3. Commands can take arguments, similarly to how functions in Python take arguments, except here they are not surrounded by parentheses. To create a directory, use the `mkdir` command with a single argument that is the name of the directory (folder) that you want to make. Create the directory `lab2`.

```
mkdir Lab2
```

4. To confirm that you have made your directory, issue the `ls` (lower case 'LS', stands for 'LiSt') command, which will list all contents of the directory where you are currently in.

```
ls
```

You should see a list with multiple items, which are the files and/or directories in your account. If done correctly, `Lab2` should be listed.

5. In graphical interface, you would double click on a folder to access that folder. In Linux, you open a directory using the command `cd`, for change directory. Enter the `Lab2` directory.

```
cd Lab2
```

6. Create an empty file using the `touch` command.

```
touch hello_world.py
```

¹Direct link: https://facultyweb.cs.wvu.edu/~wehrwes/courses/csci141_21s/labs/linuxref.pdf

7. Navigate to the Lab2 directory using the file browser on the left and click the `hello_world.py` file to open it in the repl.it editor, and add the following to the file:

```
user_name = "Scott"
print("Hello,", user_name)
```

8. Return to the terminal, and again issue the `ls` command, and you should see the just-created file listed.
9. Just as you can run a Python program using Thonny by pressing the green Run button (and in repl.it using its friendly green Run button), you can also run a program from the command line. In the terminal window (make sure you are in your Lab2 folder, which contains your Python program), run the `hello_world.py` program by invoking the python interpreter:

```
python hello_world.py
```

You should see the output of your program printed out to the terminal.

3 Command Line Arguments

We've seen how to ask the program's user to provide input using the `input` function, which returns a value of type `str` containing whatever the user entered. In this section, we'll see another way to get input from a user by having our program take *command line arguments*. Much like how the `mkdir` shell command takes an argument specifying the name of the directory you want to create, Python programs can also read arguments that are specified on the command line when they are run. We pass arguments to Python programs in the same way.

1. Try running the following:

```
python hello_world.py Jesse
```

Notice that the program runs just fine, but doesn't do anything differently from before - that's because our program currently ignores any command line arguments given to it.

2. The syntax for accessing command line arguments from inside your python program will probably look unfamiliar, because we haven't discussed these Python features yet. However, it will be convenient to use command line arguments even though we won't get around to explaining the syntax until later in the course. For now, try changing your program's code to the following:

```
import sys
user_name = sys.argv[1]
print("Hello,", user_name)
```

Don't worry about the details here - take my word for it that `sys.argv[1]` is going to be the first command line argument.

- Remember that when running a program from the command line, we're not in Python, and here arguments are specified without parentheses or commas. To run your updated program, try the same command as before:

```
python hello_world.py Jesse
```

A helpful command line tip: If you want to run the same command again, you can press the up arrow key to get the last command run to show up at the prompt; to run it again, simply press enter. You can also press the up arrow multiple times to scroll back through the history of all the commands you've run.

Try the same command but replace Jesse with a name of your choice - you should see that your program is printing whatever name you specify on the command line.

- What happens if you run your program without providing a command line argument? Try running this:

```
python hello_world.py
```

Python throws an error (Index Error: list index out of range) because we're asking for an argument that wasn't provided.

- If more than one argument is given, you may be able to guess how we access it: `sys.argv[2]` refers to the second argument specified, `sys.argv[3]` for the third, and so on. Try changing your program to say hello to two people:

```
import sys
name1 = sys.argv[1]
name2 = sys.argv[2]
print("Hello,", name1, "and", name2)
```

Then run your program with something like:

```
python hello_world.py Caroline Scott
```

Command line arguments are a little less flexible than using the `input` function because the program can't interactively ask the user for inputs. However, it can be more convenient to take input this way, especially when you're running a program over and over again while testing it: you can simply repeat the same command without having to type the input again.

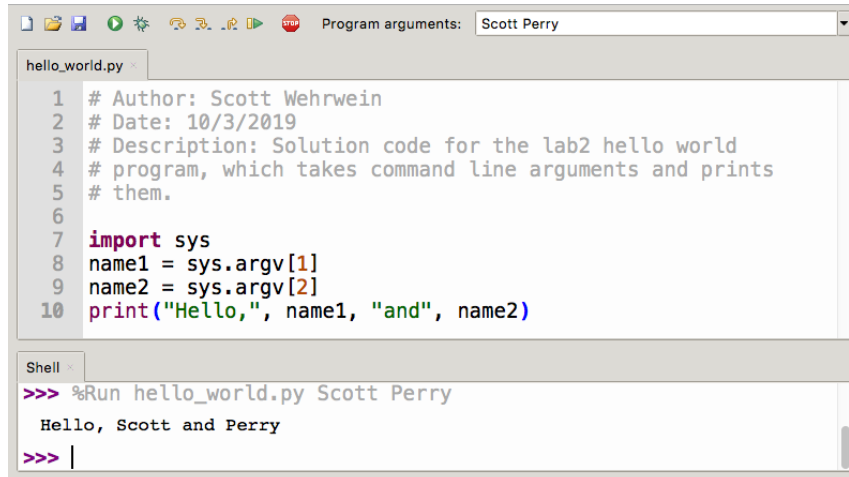
- A final question that we need to answer: we've seen how to provide command line arguments when running a python program from the terminal, but how can we accomplish the same thing in Thonny, where we run a program by simply pressing the green Run button?

Thonny lets you specify arguments to the program in a text box at the top of the window, labeled "Program Arguments". If you don't see it, you can enable it under the 'View' menu. To run the equivalent of

```
python hello_world.py Caroline Scott
```

in Thonny, type `Caroline Scott` into the Program arguments box at the top, then press the green Run button. You should see something like what's pictured in Figure 6.

For the remaining sections of this lab, you can run Python either via Thonny or via the command line.



```
hello_world.py
1 # Author: Scott Wehrwein
2 # Date: 10/3/2019
3 # Description: Solution code for the lab2 hello world
4 # program, which takes command line arguments and prints
5 # them.
6
7 import sys
8 name1 = sys.argv[1]
9 name2 = sys.argv[2]
10 print("Hello,", name1, "and", name2)

Shell
>>> %Run hello_world.py Scott Perry
Hello, Scott and Perry
>>> |
```

4 Comments, fancy printing, and debugging

Inevitably, you will write code that will have bugs, or errors, no matter how experienced of a programmer you might be. Knowing how to find and fix bugs is a critical skill for all programmers.

Using a browser such as Firefox, go to the course website:

https://facultyweb.cs.wvu.edu/~wehrwes/courses/csci141_21s/#schedule)

download the file `bad_code.py`, and save it in your Lab2 folder.

4.1 Commenting Code

Open the file `bad_code.py` in Thonny. Read over the code, paying particular attention to the comments. Proper commenting is crucial to writing good code, so that it is easy to read by you and others. In python all comments begin with the `#` character. As a general rule:

- Include your name (the author) and a short description at the top of each `.py` file
- For every few lines of code, include a descriptive comment
- Use white space (blank lines) to chunk your code into logical sections

This code has a bug and won't work, but it is a good example of commenting your code. For all the labs and assignments in this course, commenting is a portion of the rubric. Get into the habit now of commenting your code well! Not only is it helpful to you, it's a required part of each lab and assignment.

4.2 Using `print` with `sep` and `end`

Notice that this code uses the `sep` keyword argument to the `print` function. Recall from lecture that adding `sep=""` (short for "separator") after the rest of the regular arguments prevent print

from adding spaces between items that you want printed. If you use `sep=""`, the only spaces that are included in the output are the spaces that you place explicitly into the Strings. For example:

```
print("Taking", "CSCI", 141, "woo-hoo!")
```

prints the following:

```
Taking CSCI 141 woo-hoo!
```

but

```
print("Taking", "CSCI", 141, "woo-hoo!", sep="")
```

would print the following:

```
TakingCSCI141woo-hoo!
```

Recall that the `print` function defaults to " " as the separator between the arguments it prints. If you want a different separator, you can give any string to the `sep` argument, such as the empty string (`sep=""`) or a comma (`sep=","`), or any other string you'd like (e.g., `sep=" uhh, "`). Try out a few calls to `print` with different separators in the interactive Shell pane in Thonny.

Once you have a feel for the usage of `sep`, play around with the `end` keyword argument, which specifies what gets printed after all the arguments are printed and defaults to `n`, the newline character, which moves the “cursor” to the beginning of the next line so anything printed afterwards appears on the next line.

Try running a few `print` commands with `end` specified to get a feel for its behavior.

4.3 Debugging

More often than not, a program doesn't work the way I want the first time I write it. You may find that you have a similar experience, so it's very useful to get familiar with how Python behaves when something goes wrong. `bad_code.py` has a **single** intentional error. If you run the code, you should see something like the following figure:

```
>>> %Run bad_code.py
The approximate number of reindeer is 12.4
Traceback (most recent call last):
  File "/Users/wehrwes/Documents/1920F/141/repo/lab2/skel/bad_code.py", line 12, in <module>
    num_reindeer = int(approx_reindeer)
ValueError: invalid literal for int() with base 10: '12.4'
>>>
```

Look closely at the error message - it tells you on what line number to look for the error. Fix the bug (**Hint:** review the lecture slides on data type conversion functions such as `int`). You should only need to modify the line of code that says `num_reindeer = int(approx_reindeer)`. Do not not modify any other line of code, add additional code, or delete any line of code.

5 More Debugging

Download `faulty_code.py` from the course webpage and save it in your Lab2 folder. Take a close look at the code. Notice again the good commenting style. Unfortunately, `faulty_code.py` has multiple errors. Run the code and look closely at the error message. Determine the cause of the error, fix it, and try running the code again. Repeat this process until the program runs without error and the output of the program matches the output shown in Figure 1.

```
>>> %Run faulty_code.py
Intro to python programming: CSCI141
bHam zip code minus the NYC zip code is 88204
>>>
```

Figure 1: Output of the corrected `faulty_code.py` program.

6 Operands and Operators

In lecture we have discussed operands and operators. Make sure that you know the definition of both. If you do not recall, review the lecture slides.

In this section, you'll solve the following programming problem: **The period key on your keyboard is broken, but you would like to multiply two numbers with a decimal digit.**

Let's look at an example. Suppose you want to calculate 20.4×17.7 , but can only enter 204 and 177. The desired result is 361.08. If the user inputs the values 204 and 177, how can you convert them to 20.4 and 17.7? By using the modular and integer division operators! For example, 204 modulus 10 has a remainder of 4, which gives the decimal value .4, while 204 integer division 10 gives 20, which is the number before the decimal in 20.4.

The Math: Suppose that for the input values 204 and 177 you have successfully extracted the whole and decimal values (i.e., 20, 4, 17 and 7). How can you calculate the result of 20.4×17.7 ? Multiplication of decimal values requires you to sum the following four parts:

- The first integer times the second integer
- The first integer times the second decimal
- The first decimal times the second integer
- The first decimal times the second decimal

Notice that for each decimal, we also multiply in a factor of 0.1 to make sure that it is correctly

weighted in the final product. In our example, the calculation looks like this:

$$\begin{aligned} 20.4 * 17.7 &= (20 * 17) \\ &+ (20 * 7 * 0.1) \\ &+ (4 * 0.1 * 17) \\ &+ (4 * 0.1 * 7 * 0.1) \\ \\ &= 340 + 14 + 6.8 + 0.28 \\ &= 361.08 \end{aligned}$$

1. Download `broken_calculator.py` from the course webpage and save it to your Lab2 folder.
2. That file is incomplete. Only two lines of code have been written, which you are not allowed to edit. The rest are comments. Lines of code that say `# COMPLETE THE CODE` you will need to write. Read the comments for each section to get a sense of what code you need to write. Also, the number of `COMPLETE THE CODE` comments in that file is how many lines of python code I (Scott) wrote in my solution. It's okay if your solution uses fewer lines of code, or more, but each block of code should accomplish what the comment above it specifies.

For the lines of code that you write, you are only allowed to use the print function, the assignment operator, and the following mathematical operators:

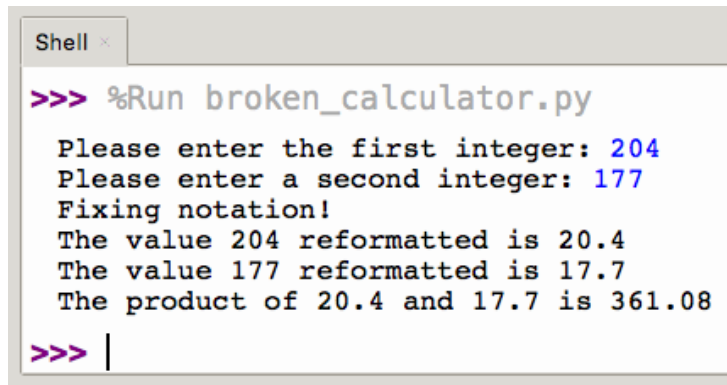
- `//`
- `%`
- `*`
- `+`

For the lines of code that you write, you cannot use `float()`, `int()`, `nor /`.

3. There are 7 parts to the code, labeled A through G. Here are hints for each of them:
 - **A:** This requires a single use of the print function
 - **B:** Use only the `//` and `%` operators. Follow the logic in the description above
 - **C:** This requires a single use of the print function
 - **D:** Do the same for the second integer as you did for the first integer (step B). Use only the `//` and `%` operators
 - **E:** The same as step C, but for the second integer
 - **F:** Use the Hint above for explanation on how to do this
 - **G:** This requires a single use of the print function.

In your python code, you CAN make use of periods, but when the program is RUN, the user CAN ONLY enter integer (non decimal) numbers.

A sample output for the completed code is shown in the following figure:



```
Shell x
>>> %Run broken_calculator.py
Please enter the first integer: 204
Please enter a second integer: 177
Fixing notation!
The value 204 reformatted is 20.4
The value 177 reformatted is 17.7
The product of 20.4 and 17.7 is 361.08
>>> |
```

7 Extra Credit: Debugging tutorial

Thonny has some excellent built in debugging tools, but if you look under the `help` menu and read the `Using Debuggers` section, the help guide is not super easy to understand.

For this extra credit lab work, create a better tutorial for the Thonny debugging tools for a student who is no further than 2nd week of CSCI 141. You should cover the basics (step over, step into, step out, resume, break points) and you can create this as a document or as a video. Submit your creation to the Lab 2 Extra Credit Assignment on Canvas for up to 5 extra credit points (1 for effort, 2 for clarity, and 2 for completeness). The best ones will be shared with the class!

Submission

Upload the following files to Canvas:

- Your fixed `faulty_code.py` code file
- The completed `broken_calculator.py` file that can reproduce the output shown in the above figure

Rubric

<code>faulty_code.py</code> has been fixed, and is properly commented	3
<code>broken_calculator.py</code> uses only <code>%</code> , <code>//</code> , <code>+</code> and <code>*</code> operators for the lines of code that you have written	3
<code>broken_calculator.py</code> has no syntax errors	3
<code>broken_calculator.py</code> produces the correct output	3
<code>broken_calculator.py</code> is properly commented	3
Total	15 points