LAB 7: Key Concepts

❑ Paths (Absolute vs Relative)

Before opening a file, you need to understand where the file is currently on your computer. Before you get to the file, you need a **path** to where the file is from where ever you are starting from.

There are 2 different types of paths we will work with in this lab:

- **Absolute paths**
- **Relative paths**

The difference between the 2 is as follows:

- **Absolute path** is "specifying the location of a file or directory" on your system from the top most directory of your system, known as the root folder of your system. For Windows, this is you *C* drive. For Mac, you will have a slash */*.
- **Relative path** is "the path related to the present working directory". This path starts from where-ever you are running your program from.

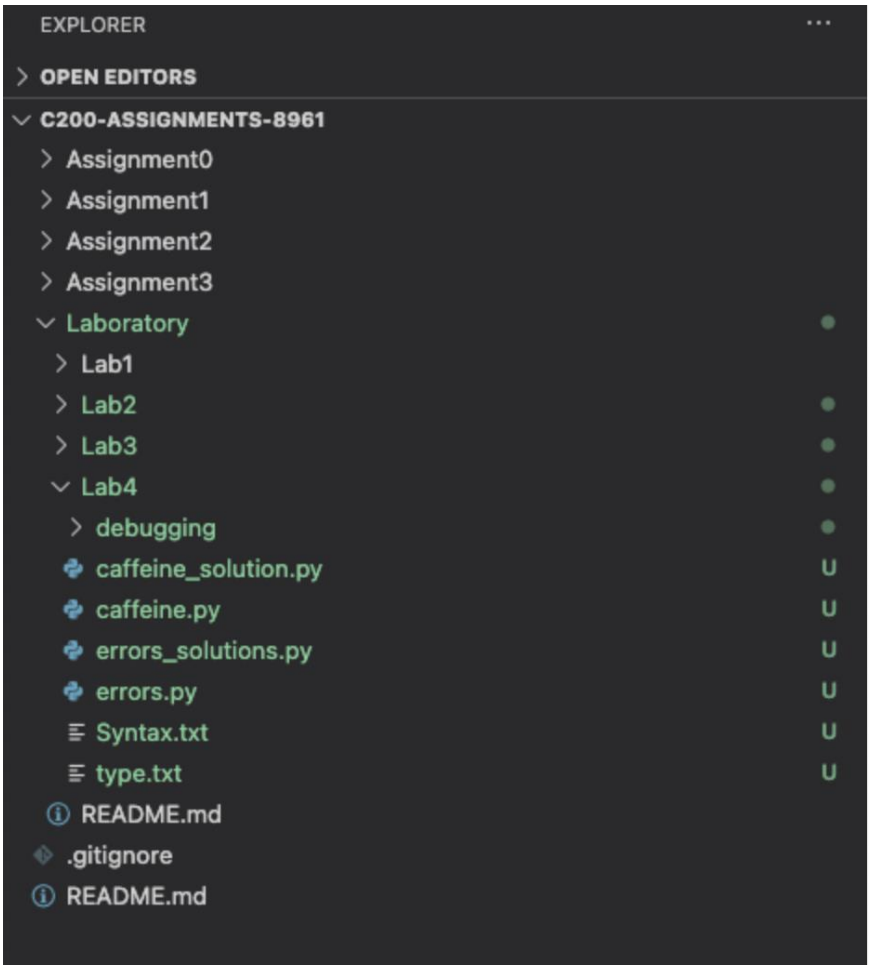
➢ Why is there a difference between the two?

Absolute and relative paths have different impacts per system. If you have a file from your current computer, say C:\Users\brown\Repo\Lecture\loops.png and reference it in your program that is at C:\Users\brown\Repo\program.py, the program will know where to look on **this** computer. Say I send my program to another person, including the Repo folder. If they try running their program at /usr/josh/Repo/program.py, they will get an error because the program will look specifically for C:\Users\brown\Repo\Lecture\loops.png.

To avoid this problem, we use relative paths. The path is relative to where the program is run from. The way we set up Visual Studio Code, all of your programs will run from the top of your repository (where C200-Assignments-username is the present working directory).

All assignments will use *relative* paths. If you use an absolute path, it will cause a problem with our grading and will not be graded.

An example of the path can be shown in the image (Is included on the Canvas page for the lab section):



Looking at this inside of Visual Studio Code, we can see that we are currently at the folder C200-Assignments-8961. All files that we access will be **relative** to this folder.

The folders that are visible at the *top level* are:

- Assignment0
- Assignment1
- Assignment2
- Assignment3
- Assignment4
- Laboratory

These are the files that are visible at the *top level*:

- .gitignore
- README.md

So, if we want to reference a file at the top of this folder, the path would be as simple as README.md.

If we want to access a file at a different location, the path would be Laboratory/Lab4/Syntax.txt. When writing programs, you want to make file access relative to the same folder. Files will be accessed based on the **current working directory**. In the lab, we will show a command that can help you figure out where you are running the Python program from in your directory.

For the people that are not sure, "a good way to remember the difference between a backslash and a forward slash is that a backslash leans backwards"

Spelling is essential for files and paths. And your files and paths shouldn't have "space" in them.

❑ Files

- Working with a file in Python, the key words you will here is read and write. We will explain the difference. You will see a code example in the *Code Demonstration*.
- With the knowledge we are giving you in this class, you are limited to either *reading* or *writing* but not both at the same time.
- When you read from a file, you have multiple options to read from. The options we are going to teach you are reading a file line by line and, reading the entire file at once.
- We have been using the word *line*. How do we know what a line is? Behind the scenes, there is a special character \n. The \n represents a new line. You will also need to utilize this when **writing** to a file.
- To do any of these operations on a file, you do **not** need to import any additional modules, as this is built-in to Python.
- If you are reading or writing from a file, you will be dealing with strings. When you read from a file, even if it is just a single number, the function reading the file will return a string. You must remember that when trying to write a program to interpret the information from the file. If you forget and try doing an operation that is not a string (such as adding a number to the value in the file) without convert the contents of the file to an integer, you will get a type error.
- When reading or writing files, you have to first open the file. But like a door, you also need to **close** the file. In python, if you forget closing a file, it doesn't cause too much of a problem. But if you do this in other languages, the languages have more errors (such as C and memory management). If you do forget to close a file in python but you try to open it in the same running program, there will be an error (since you can't open a file a second time if it is still open).

- **List Comprehensions** (i.e., **inline for-loops**): You can think of a list comprehension as an "inline for loop" that builds up a list.

```
>>> # List Comp
>>> [2*i for i in range(4)]
[0, 2, 4, 6]
>>>
>>> # We can do multiple for-loops
>>> [(i,j) for i in range(4) for j in range(3)]
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2), (3, 0), (3, 1), (3, 2)]
>>>
>>> # We can also filter the results by an if-statement at the end:
>>> [i for i in range(10) if i % 3 == 0]
[0, 3, 6, 9]
```

- **Part 1**

The students should download all files from *Canvas* and put them in the following path: Laboratory/Lab7/

The files should be:

- filing.py
- blank.txt
- testing.data
- calculation.txt

File Reading and Writing

We will start with filing.py. At the very top of the file is a function we provided. This function has a command from the os module that allows us to see the current working directory. They don't need the whole function, just the line os.getcwd() if they want to understand where they are running their code.

The path we see here is an **absolute** path. In the program, we will use **relative** paths, that are relative from the current location of the program.

Reading

After putting all the required files inside Laboratory/Lab7/, we can start coding in filing.py and in function readingEx1().

Inside of readingEx1() you will type in the following:

```
with open("Laboratory/Lab8/blank.txt", "r") as someFile:
    someFile = open("Laboratory/Lab8/blank.txt", "r")
    contents = someFile.read()
    return contents
```

Emphasize the following:

- If you don't include the mode of reading (r or w ...), it will default to reading a file
- When you read from a file (as stated earlier), it stores all the information as type str (string)
- As soon as you are done with the file, you should close it to free up the memory taken by the I/O stream. The with open() as ...idiom automatically closes the file when we exit the block.

Your output should be:

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1
2
3
4
5*EOF*
__________________
```

A note about the output:

- The *EOF* shows where the end of the file is. It doesn't go to the new line, there are only 5 lines in the file and, the end of the file ends at the line with the number 5.

Move to the next function, readingEx2(). The function will take advantage of readlines. We can infer that the code will mostly be the same, except one small change.

```
with open("Laboratory/Lab8/blank.txt", "r") as someFile:
    open("Laboratory/Lab8/blank.txt", "r")
    contents = someFile.readlines()
    ourPrint(contents)
```

The output will be:

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
['1\n', '2\n', '3\n', '4\n', '5']*EOF*
--------------------
```

Here, we can see that readlines gives back a list of strings. For each newline, we will explicitly see the \n here (since the print does not print everything inside of the list, just the outer list). Notice the last line does not have a \n.

The function readlines is a simpler way of splitting the lines. But you need to make sure to strip out \n if you want to do any further processing.

Writing

Let's now look at a few examples of file writing, because we may want to store program's output in a file.

Due to time constraint, we are just going to be working with write. There is also an option of writelines that students can learn later, but essentially it takes in a list of strings and writes to the file (however, does not write individual lines).

Inside of writeEx1(), we will put the following code:

```
stuff = ["a", "b", "c", "d", "e", "f"]
with open("Laboratory/wrong.txt", "w") as fileToWrite:
    for s in stuff:
        fileToWrite.write(s)
```

Now, after writing the file. Is it in the *Lab8* folder? It is not, we can see the path we wrote to is **NOT** in the *Lab7* folder. Let's fix the code and try again. (you can delete or leave wrong.txt in the wrong place, but it is to demonstrate how easy it is to mess up the path, we will only be looking for wrong.txt in Laboratory/Lab7/wrong.txt).

```
fileToWrite = open("Laboratory/Lab8/wrong.txt", "w")
```

Run the file

What we see here:

- The write function is writing a single string.
- We need to open with w so Python knows that we want to write to a file. You can't use the write function if you are reading (r in open).

We also notice that all the items are on the same line. To fix this, they need to add a line inside the for loop.

```
fileToWrite.write(s + "\n")
```

Run the file

Now, this part is important, **commit and push** (as we want to demonstrate something).

After you save, **manually** edit Laboratory/Lab7/wrong.txt. Type literal gibberish and **save** the file. Then run filing.py again. you will see the changes are gone.

The mode w (write) will either create a new file (even if the file exists), basically overwriting an existing file. This is not useful if we don't want to erase the contents of the file.

We will now move to writeEx2(). The code will look the same as the previous function, but we will use the mode a (append).

```
with open("Laboratory/Lab8/wrong.txt", "a") as fileToWrite:
    for s in range(4):
        fileToWrite.write("more\n")
```

We will now see that the file will have new lines in it, and in the function writeEx2 we did not write the original list. Since the contents were already added, we just **appended**.

Part 2

Time to put these to list comprehension to the test! For the code demo, we will be walking through:

- how to rewrite a for-loop as a list comprehension
- how to do the same thing using only map and lambdas

Open LC.py and follow LC_solution.py. We only have time for converting the one function. But if you have extra time, there is a challenge for the students: How to do the map-variation without the assumption that the list elements are distinct.

Notes:

Emphasize the following points:

- When you open a file, close it after you are done with file reading/writing.
- When you read from a file, you will read a string (even if it looks like an integer or something else).
- String formatting comes very handy while reading/writing files.