CMSC 201 Fall 2015
Lab 08 – Strings and File I/O

Assignment: Lab 08 – Strings and File I/O
Due Date: During discussion, October 19th through October 22nd
Value: 1% of final grade

Part 1: File Input

Using files as input is a much quicker and easier way to get information from the user, especially for large amounts of data. Rather than having the user enter everything by hand, we can **read in the data from a file**.

To open a file for reading, we use the following command:

```
myInputFile = open("theFile.txt", "r")
```

This line of code does three things:

1. It opens the file `theFile.txt`
2. The file is opened for **reading** (`"r"`) – as opposed to writing
3. The opened file is assigned to the variable `myInputFile`

Once we have opened a file and assigned it to a variable, we can use that variable to access the file. There are four different ways to read in a file.

1. Read the entire file in as one enormous string
   ```
   myInputFile.read()
   ```
2. Read the file in as a list of strings (each line being a single string)
   ```
   myInputFile.readlines()
   ```
3. Read in a single line of the file
   ```
   myInputFile.readline()
   ```
4. Iterate over the file using a `for` loop, reading in a line each loop
   ```
   for singleLine in myInputFile:
       # singleLine contains a line from the file
   ```

Often, if we want to extract or examine data from a file, the last option (using a **`for`** loop to iterate over the lines of the file) makes the most sense to use.

Below, you can see an example where we read in from a file, printing only those lines that are *exactly* 36 characters long.

```
inputFile = open("road.txt") # Robert Frost's poem
for line in inputFile:
    line = line.strip()   # remove the newline (and
                          # any other whitespace)
    if len(line) == 36:   # choose the lines to print
        print(line)
inputFile.close()
```

When the file “road.txt” contains the poem “The Road not Taken” by Robert Frost, here is what the code above outputs:

```
Two roads diverged in a yellow wood,
To where it bent in the undergrowth;
And having perhaps the better claim,
Though as for that the passing there
Had worn them really about the same,
In leaves no step had trodden black.
Yet knowing how way leads on to way,
Two roads diverged in a wood, and I—
```

Part 2: String Manipulation

This is fine, but often we want to look at the contents of a line, and make a decision based on that, rather than on something trivial like the line length.

For example, we may have a file that contains information about our employees and how many hours they worked this week. Using this information, we want to be able to determine which employees are full-time (work 30 hours or more) and which are part-time.

If we know the format of the file we are reading in, we can take advantage of the **`split()`** function to assign each "word" in a line to individual variables.

If we take a look at the **`totalHours.txt`** file, we can see that each line is formatted the same: employee id, employee name, and the total hours worked that week. Since we know the format, we can directly assign each piece to a separate variable, and use those variables to help decide which employees are full-time.

```
totalHours.txt
123 Susan 18.5
456 Brad 35.0
789 Jenn 39.5
101 Thom 28.6
```

One important thing to remember is that all of these variables will be strings to start off – so if we want to use them as integers or floats, we will need to first cast them to be that type.

```
workerHours = open("hours.txt")
for line in workerHours:
    # directly assign each "word" to a variable
    id, name, hours = line.split()
    # remember to cast to another type if needed
    if ( float(hours) >= 30):
            print(name, "is a full-time employee")
    else:
            print(name, "is only a part-time worker")

# don't forget to close the file!
workerHours.close()
```

The code on the previous page will give us the following output:

```
Susan is only a part-time worker
Brad is a full-time employee
Jenn is a full-time employee
Thom is only a part-time worker
```

By default, the **`split()`** function uses whitespace (spaces, newlines, tabs, etc.) as the delimiter, *i.e.*, the boundary between what constitutes a “word.” However, we can also give it a specific character (or characters) to split on. Here’s an example from class:

```
nonsense = "nutty otters making lattes"
nonsense.split("tt")
# which will output this list of strings:
# ['nu', 'y o', 'ers making la', 'es']
```

This is a bit of a silly example – normally, if we are not splitting on the whitespace, we are likely splitting on some other sort of separator character. Using commas, semicolons, and underscores are all common choices, as can be seen in the example code below:

```
famous = "Bill Nye, the science guy"
famous.split(",")
# which will output this list of strings:
['Bill Nye', ' the science guy']
```

Part 3: File Output

Printing your output to a file instead of the terminal is a great way to store information for future use, or when there is so much information it would overwhelm the user.

To open a file for writing, we use the following command:

```
myOutputFile = open("theFile.txt", "w")
```

This line of code does three things:

1. It opens the file `theFile.txt`
2. The file is opened for **writing** (`"w"`) – as opposed to reading
3. The opened file is assigned to the variable `myOutputFile`

Notice that the only difference between using `open()` to open a file for writing versus for reading is the letter given to select the access mode: "`w`" for write, and "`r`" for read.
When a file is opened for writing, the contents of the file are wiped – in other words, the information the file used to contain has been deleted.

Once we have opened a file and assigned it to a variable, we can use that variable to write to the file, using the appropriately-named `write()` function. Although using `write()` may seem similar to `print()`, the two actually work and behave very differently.

The main difference is that `write()` only works when given a single string. It can handle two strings that are concatenated together (since they evaluate to a single string), but cannot handle multiple strings or variables separate by commas.
Additionally, when non-string variables are concatenated onto a string, they must first be cast to a string.

```
# this will NOT work
myOutputFile.write(name, "is now", age)
# this will work
myOutputFile.write(name + " is now " + str(age))
```

The other difference is that **`write()`** does not automatically add a newline to the end of each printed line. Instead, you must include the escape sequence for a newline ("**`\n`**") at the end of the string you are writing. Both of the following ways of including a newline will work:

```
# at the end of the last string
myOutputFile.write("Line one\n")
# concatenated to the end of the string
myOutputFile.write("Line " + str(2) + "\n")
```

Part 4: Using Multiple Files

One very important thing that Python does is it allows us to have more than one file open at a time. The different files can be opened for reading, writing, or appending! They can even be opened in different modes, like one file opened for write and one opened for read.
This makes it very easy to directly write important information from one file to another!

For example, if we wanted to read in a file and save a copy of it, but with the copy in all capital letters, the following code would accomplish that:

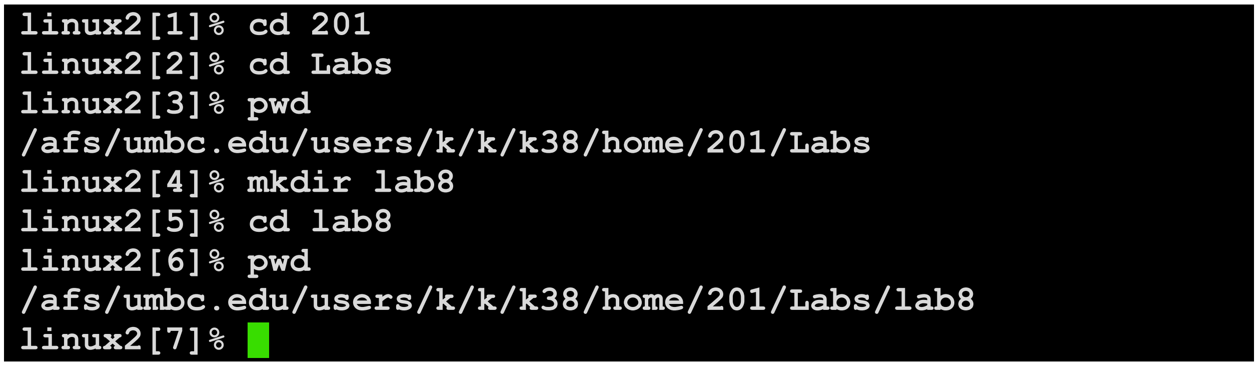
```
inputFile  = open("road.txt", "r")
outputFile = open("ROAD.txt", "w")
for line in inputFile:            # read in the input
    capsLine = line.upper()       # convert to all caps
    outputFile.write(capsLine)    # print out the output
inputFile.close()
outputFile.close()
```

If we take a look at the file using the "**`more`**" command from the terminal, we can see that everything has been capitalized now:

```
bash-4.1$ more ROAD.txt
TWO ROADS DIVERGED IN A YELLOW WOOD,
AND SORRY I COULD NOT TRAVEL BOTH
[and so on]
```

Part 5A: Writing Your Program

After logging into GL, navigate to the **`Labs`** folder inside your **`201`** folder. Create a folder there called **`lab8`**, and go inside the newly created **`lab8`** directory.



```
linux2[1]% cd 201
linux2[2]% cd Labs
linux2[3]% pwd
/afs/umbc.edu/users/k/k/k38/home/201/Labs
linux2[4]% mkdir lab8
linux2[5]% cd lab8
linux2[6]% pwd
/afs/umbc.edu/users/k/k/k38/home/201/Labs/lab8
linux2[7]%
```

To open the file for editing, type

```
    emacs dealership.py &
```

and hit enter. (The ampersand at the end of the line is important – without it, your terminal will "freeze" until you close the emacs window. **Do not include the ampersand if you are not on a lab computer.**)

The first thing you should do in your new file is create and fill out the comment header block at the top of your file. Here is a template:

```
# File:         dealership.py
# Author:       YOUR NAME
# Date:         TODAY'S DATE
# Section:      YOUR SECTION NUMBER
# E-mail:       USERNAME@umbc.edu
# Description:  YOUR DESCRIPTION GOES HERE AND HERE
#               YOUR DESCRIPTION CONTINUED SOME MORE
```

Part 5B below will explain the assignment. Then you can start writing your code for the lab, following the instructions in Parts 5C and 5D.

Part 5B: Assignment – A Used Car Lot

For Lab 8, you own a used car lot, and every morning one of your employees types up a text file called "`cars.txt`" that contains information about each of the cars you have on the lot that morning. Each line of the file is structured as seen below, where the items are separated by a comma:

```
    car name,car color,# of doors,# of cup holders,price
```

A potential customer walks onto your lot one morning. They are looking for a **red car with 4 doors that costs less than $30,000**. Using Python, File I/O and string manipulation, figure out if you have any cars that match the criteria specified. (You do not need to get any input from the user – you can hard code their criteria directly into your program.)

However, the customer can't see your screen, so they've requested that you email them the list as an attached file. Create a file called "`results.txt`" that contains the name and price of each car that matches their criteria.

You will be coding Lab 8 in an incremental manner – in other words, you will code up one piece of the lab and test that it works **before** moving on to the next piece. Incremental development is a very effective way of tackling a problem, in part because it is easier to figure out where an error occurs when you are only working on a small part of the code at a time.

Part 5C: Reading in the `cars.txt` File

Before you begin, download the **`cars.txt`** file by running this command in your **`lab8`** directory:

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

After you have the file in your directory, the first piece of code you should write will open the file "**`cars.txt`**" for reading. (Don't forget to also write the code to close the file when you are done!)

You should then write code that reads in the contents of the file. You should use one of the methods described in Part 1.

Once you can correctly read in the file, you should write code that will handle parsing the data present on each line. As a reminder, it looks like this:

```
car name,car color,# of doors,# of cup holders,price
```

"Parsing" data means to analyze the data, or to break it into its respective pieces. Since you already know the format of the data in each line, you should be able to parse it relatively easily.

(There are hints on the next page if you need them.)

Try to solve Part 5C on your own before you turn to these hints!

Getting an error that looks something like the following?

```
FileNotFoundError: [Errno 2] No such file or directory:
```

Make sure that you have downloaded the **`cars.txt`** file, and that you have spelled the filename correctly when you attempt to open it.

Having trouble reading in the lines individually?
As stated in Part 1, if you want to extract or examine data from a file, using a **`for`** loop to iterate over the lines of the file makes the most sense. See Part 1 for an example of the syntax.

Not sure how to parse the data on each line?
There are two key pieces of information in Part 2 that will help you:

1. If you know the format of a file, you can assign each piece to a variable:
   ```
   id, name, hours = line.split()
   ```

2. Although **`split()`** uses any white space as its default delimiter, you can give it a different character to split on:
   ```
   line.split(";")
   ```

Is your code not doing what it should, and you're having trouble figuring out exactly where it's going wrong?
Try having your code print out what it's seeing/doing at each point. For example, if you're having trouble assigning the parts to variables, try having it print out each variable as it is read in and assigned – you may spot the problem that way.

Part 5D: Selecting (and Saving) the Correct Car

Do not move on to this part until your program can read in the **`cars.txt`** file!

Now that you have one piece of your program working, we can focus on the next task. You've read in the list of available cars, but now you need to select only those cars that fit the customer's requirements. As a reminder, your customer is looking for a **red car with 4 doors that costs less than $30,000**.

Using the data that you read in from the file in Part 5C, find the cars that match the given requirements. Once you have found one of these cars, write its name and price to the output file "**`results.txt`**".

You can check the contents of **`results.txt`** either by opening it in emacs, or by using the following command to see the file in the terminal:

```
more results.txt
```

Here is an example of what a successful run looks like. (Notice that because we are using input and output files, the Python program doesn't actually print anything to the terminal, and there is no user input.)

```
linux2[6]% /usr/bin/scl enable python33 bash
bash-4.1$ python dealership.py

bash-4.1$ more results.txt
Ford Fiesta --- $19653
Hyundai Elantra --- $18032
Honda Fit --- $16530
```

(There are hints on the next page if you need them.)

Try to solve Part 5D on your own before you turn to these hints!

Does your results.txt file come out blank?

The most likely cause of this is that you are failing to find the matching cars. Remember, variables like the number of doors and the car price are read in as **strings**, and won't match with an integer. In other words, `4 == "4"` will evaluate to False. Make sure that you cast the variables before comparing them.

Still having trouble selecting the correct cars?

If you are still having trouble, try printing your output to the screen instead of the `results.txt` file. This will help you pinpoint where the problem lies. If you're not printing the correct cars out, the problem is likely with your code that checks for the correct car specifications. If you are printing the correct cars out to the screen, the problem is likely with your file output code.

Getting an error that looks something like the following?

```
    NameError: global name 'resultsFile' is not defined
```

Make sure that you opened the output file before you started writing to it! That is how we initialize the variable for Python – we have to tell it what filename it's supposed to output to before we can use `write()` on it.

Part 6: Completing Your Lab

To test your program, first enable Python 3, then run **`dealership.py`**. You will need to view the **`results.txt`** file to check that everything worked.

```
linux2[7]% /usr/bin/scl enable python33 bash
bash-4.1$ python dealership.py

bash-4.1$ more results.txt
Ford Fiesta --- $19653
Hyundai Elantra --- $18032
Honda Fit --- $16530
```

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!