FILE INPUT

Most programs that you use interact with other files on your computer. Having some easy way to interact with files (for reading from and writing to) is essential. Fortunately, Python makes it about as easy as possible! You can make simple text files in ***Notepad*** or even in ***Wing*** by choosing '*Save as*'… '*Save as type*'… '*Plain text*'.

The following code opens a file from the user's computer in a specific location, reads every line in the file, and prints them:

```
in_file = open(r"G:\My Drive\ICS3U\numbers.txt")
for line in in_file:
    print(line)
```

← *Note the 'r' before the quote!*

Note: You can use **`in_file = open("numbers.txt")`** if the file is in the same directory as your current python file. This method is often better since if you move the program to a different computer (like when you submit), the other user's directory structure may be different.

Here the `for`-loop *iterates* over every line in the file. The iterator variable called **`line`** takes on the value of the first line, then the second, etc. Each line in the file is read in as a **`str`** (string). This means that we can use all the string methods and slicing for manipulating the values. If the file has numbers in it, you can use `int()` and `float()` to convert the values read in to numeric types. The loop will automatically stop when the loop has read all the lines in the file.

The output of the above example is a little strange. Notice that there is an extra blank line after each line in the file that is printed. This is because whenever you use a print statement, it always puts a 'newline' (like the 'enter' key) at the end of what it prints. You know this. What you don't realize is that when Python reads a line from a file, it reads a hidden 'newline' at the end of each line as well (this is one of those **`\n`** characters we used when building strings!). This `\n` becomes part of what it prints. So it prints the line including the `\n` (which means go to the next line) and then adds another new line by default as part of the print statement.

One way around this is to cut off the `\n` at the end of each line (if it exists). There is no `\n` on the last line in the file. If you review the list of string methods that were discussed earlier, you might see the usefulness of the **`strip()`** method. One version of this method says that it will strip off all of the whitespace off each end. The other version allows you to define the characters that you would like to strip off of each end. `\n` is a character!

```
for line in in_file:
    line = line.strip("\n")
    print(line)
```

→ Now this will print each line without the extra newline.

Reading files is not just about re-printing them. We generally want to use the data read from files in some other way. Removing the `\n` generally leaves us with just the text we want.

Let's consider an example of using the data in the file for something else. Say you want to know how many characters (letters, symbols, spaces, numbers, etc.) are in a file. You are not interested in treating `\n` as a character, so you don't want to count those. Recall that the `len()` function returned to you the length of the string it was given as an argument. This is what we want, but we want to add up the `len()` of each line.

```
in_file = open("somefile.txt")      ← a file in the same directory
total = 0
for line in in_file:
     line = line.strip("\n")  # take the \n off of line
     total += len(line)
print(f"There are {total} characters in the file.")
```

You may also want to apply an actual string method to all the lines in the file:

```
in_file = open("somefile.txt")
upper_file = ""
for line in in_file:
     line = line.strip("\n")  # take the \n off of line
     upper_file += line.upper() #builds up all lines as uppercase
print(upper_file)
```

This example is a little interesting. Notice the use of building up a string. However, since we stripped off the `\n`'s, they are not being added to the built-up string. This means that the output won't be in the same format as the input file, since all line breaks were removed. You could either put them back in or never strip them in the first place!

Reading one line at a time

The `for`-loop structure you have seen so far will read through the ***entire*** file. You can also read one line at a time using the `readline()` method:

```
in_file = open("somefile.txt")
first = in_file.readline() #reads the first line
next = in_file.readline()  #reads the second line
```

The file will keep track of where it is in the file, so you don't have to worry about telling it what specific line to read. The problem here is that you could have read the last line, and you won't know it! When you attempt to read beyond the last line, `readline()` returns to you the empty string `""`. That is not a huge issue, because you can add a condition checking to see if the line you read is the empty string. However, what if you had a blank line in the middle of your file? In that case, `readline()` would return the empty string (with a `\n`), and that may confuse your program!

The lesson here is that you can read individual lines (and often need to), but you better be sure you know exactly what the structure of the file is.

Reading in the entire file at once

You can read the entire contents of the file into a single (possibly *very* large) string by using the `read()` method. This single statement will store the entire file contents, including all line separators (`\n`) in a single string variable. While convenient, it can also be difficult to work with that one very large string. A sample: `all_contents = in_file.read()`

Exercises:

1. Write a function called **`file_average()`** that takes the name/path of a file which contains one `float` value per line (no blank lines anywhere!) and returns the average of the numbers in the file. Make a file with at least 5 lines/numbers to test with.

2. Write a function called **`count_letter_in_file()`** that takes two arguments: the name of a text file and a single character to look for (e.g., "r"). The function will return the number of occurrences of that letter (in either casing) in the file.

3. Write a function called **`top_student()`** that takes the name/path of a file where each line in the file has a student name and that student's average (e.g. `Jon Smith 76.9`) separated by spaces. There will be many lines in the file. Each student could have a different number of parts to their name (e.g. `Guido van Rossum`).
   Return the name of the student with the highest average.

4. Write a function called **`count_code_lines()`** that takes the name/path of a *Python* file as an argument. The function will return how many lines of '*code*' are in the file. 'Code' doesn't include blank lines, or comment lines starting with '#'. Don't worry about handling triple-quote comments.

5. Write a function called **`ids_with_digits()`** that takes two arguments: the name/path of a file containing student ID numbers (one per line), and a string of digits to look for in each ID. The function will return a string showing all ID numbers that contain each of the digits in the given string (one ID per line). For example, the ID "346128223" contains all the digits in "4261" but does *not* contain all the digits in"52". The rci_ids.txt file has been provided to test with.