Laboratory 7: Reformatting Files

In this laboratory you will learn or apply your learning to opening a file and changing its contents to a different format.

1. Opening and writing to files

Before we read or write a file, we need to open the file. We do so with the open statement. In

```
outfile = open(filename, "w", encoding = "utf-8")
```

we open a file with the name filename and call the result outfile. The open call has two additional components. The "w" means that we are open the file for writing. If we leave it out or replace it with a "r", then we open up the file for reading. (There is another class of modes, namely binary versus text, but we are only going to use text mode in this class and there is no need to specify it since it is the default). The last argument from open can also be deleted, it specifies how the text is going to be encoded, e.g. ASCII, iso8859_5 (Cyrillic), iso8859_8 (Hebrew), or utf-8, the de facto standard for webpages and cloud-data. The result of open is a **file pointer** through which we programmatically gain access to the file. Here, the file pointer is stored in a variable called "outfile".

After we use a file, we have to close it so that other programs have access to it. The statement is

```
outfile.close()
```

We can save ourselves the two statements by using the new with-construct. We just say:

```
with open(filename, "w", encoding = "utf-8") as outfile:
```

What follows is an indented block during which we can access the file and at the end of which outfile is closed automatically.

To write to a file, we can use the print statement with an additional, named argument. If we want to write variables x, y, and z to the file, we just say (inside the with block):

```
with open(filename, "w", encoding = "utf-8") as outfile:
      print(x, y, z, file = outfile)
```

Task 1: Select a directory (folder in Mac and MS-speak) for your program that you are going to write. Inside this directory, create a **new** directory called Data. (How you do this depends on your OS and how you access your file system.)

Task 2: Write a function that writes a sentence to a file called "example.txt" in the directory Data that you just created. To do so, you need to give the relative path to the file, i.e. you need to use

"Data/example.txt"

if you are using MacOS, Linux, or any other Unix flavor, and

"Data\\example.txt"

if you are using a WindowsOS. The first of the double backslashes in the Windows version is an escape character. If you watched the presentation on f-strings, then you know that you can also use

r"Data\example.txt"

for the filename which is a raw string.

Task 3: Modify your function to `def writeData(filename, nr_points)` so that it write nr_points lines consisting of three numbers.
The first number is in $\{10 + i + \epsilon \,|\, i \in \{0,\ldots,n-1\}\}$ where ϵ is a simulated error, the second number is the logarithm of the first plus a simulated error, and the third one is the square of the first one plus a simulated error. To simulate an error, we import the random module (`import random`) in the first line and then use `random.gauss(0, 0.2)`. This function returns a random floating point somewhere close to 0. The first parameter is the mean, the second is the standard deviation, if you know statistics.

At this point, in your Data directory, you should see a text file example.txt that contains a number of lines each of which contains three floating point numbers.

2. Processing Files

In order to process a file, we have to open it for reading. We can simultaneously open two files within a with-statement by chaining them, separated via comments:

```
with open(filename) as infile, open(newfilename, "w",) as outfile:
```

In this example, we open up infile for reading and outfile for writing. Notice that there is not mode given in the first open, so that it defaults to reading mode. Also, we do not specify encodings, so that the default 'utf-8' is used.

There are many possibilities for reading from a file. Since we are only using text-files, the best method for us is just to read lines. We do so with a for loop as in

```
for line in infile:
```

Task 4: Write a function that opens up the file in the Data directory and prints out all of its lines.

You will notice that each line is separated from the next line by an empty line. This is because the line ends in a newline character and the print statement adds another one. To remove white-space characters such as the newline from a line, we use the strip method.

```
line.strip()
```

is a new line without these characters. If we give a string argument to strip, then characters that are in the string are removed from the beginning and the end of the line.

Task 5: Modify your function such that it prints out the one and only file in Data line by line without an intervening new line.

In order to process a line, we have to divide it into constituent parts. We can do so with the split method. The split method separates a string into components. The argument of the split method are the letters on which we split. Its output is a list of components, i.e. a list of strings. If there is no argument, then split splits on white spaces such as the space or the tab.

Task 6: Modify your function such that it prints out a list of the components for each line.

Task 7: Modify your function such that it prints out a list of the three numbers as floating point numbers.

3. Formatting output

In order to rewrite each line, we can use either f-strings (see the short presentation) or we can use the format method. If we need more control, then we have to use the format method. The format method consists of a blueprint, followed by the method invocation, which has as arguments all the data that is to be formatted. For example, in

```
"{:6.3f} | {:6.3f} | {:6.3f}".format(x, y, z)
```

creates a string that consists of three numbers stored in variables x, y, and z separated by a space, a vertical bar, and another space. The braces represents the fields where the variable are being stored. Inside the braces, we have a colon, followed by more detailed formatting instructions. Here, they say that each field is 6 characters long, that there are 3 digits of precision, and that they are to be printed as normal floating point numbers.

Task 8: Modify the previous function to print out the numbers in the file in directory Data separated by a comma and a space. For example, one of my lines is

```
10.120,  2.316, 102.531
```

Task 9: Write a function that changes a string such as a file name "Data/data.txt" to "Data/data.csv". Your function could for example use a slice and concatenate ".txt".

Task 10: Change your function so that it rewrites all the data in filename to a similarly named ".csv" file. Then open the 'csv' file through the file explorer / finder. The abbreviation 'csv' stands for comma-separated values. Your OS is set up to associate applications to files based on the three-letter extension. If you have a spreadsheet application installed, it will open up in the spread sheet.

4. Getting a list of all files in a directory

In order to get all files in a directory, we can import the module os and then use os.listdir. os.listdir takes as argument the name of a directory and returns a list of strings, each string being the name of a file in the directory.

Task 11: Change your previous function so that it creates 10 files names f1.txt, f2.txt, … , f10.txt. You can use a loop to create all the file names.

Task 12: Create a function change_all(dirname) that lists all the files in the directory that have the extension '.txt'. You will find the string method ".endswith( )" very useful.

Task 13: Modify the previous function so that it changes the data in all '.txt' files to a comma-separated value file using the second function that you developed.

Task 14: Write a function that takes all '.txt' files in a directory (whose name is the sole argument of the function) and change it into a 'csv' file with a slightly different names where all numbers are now separated by a tab.