Homework 7 Instructions

The zip download file has, besides this file and skeleton files, several data files that you can use for testing/validating your programs as described below.

You will be writing 5 separate Python modules/files for this homework assignment. The skeleton files are included in the homework zip file. **Please put your name in a comment at the top of *each* program file.** The 5 modules are: `file_stats.py`, `re_order.py`, `tabs_to_commas.py`, `re_sort.py` and `country_convert.py`. Zip these 5 `.py` files together into **one** `.zip` file as described in the *About Homework* document. Please just zip the files, **not** a folder, thanks! **Do not include data/results files; only the `.py` files.**

Please read and follow all the directions carefully! This assignment is a bit more complicated than others. But if you think about what you are doing, and what should be happening, it should become clear how to proceed. It also helps a lot to read the instructions **carefully**. *Some of the programs require both input and output filenames to be given on the command line; some require you to create the output filename based on the input filename.*

Please email me with any questions you have! Especially if you have worked on something for an hour or two and are stuck. Don't wait until the last minute!

A few TLDR notes:

- Where there is a line `pass` in the files, *that is where you put the code*. Remove the lines with `pass` and replace with the code.
- **If you have a Windows computer:** Be sure to use `newline=""` in the file open statements.
- **Please** assume that the input to your files/functions is always valid.
- In each of the modules, you should only open the files once. **Always** use the `with` context manager.
- **Pay careful attention to the requirements for each program.** Most have requirements as to how it works on the command line *in addition to how it works in the REPL.* Some require an input filename alone, some an input *and* output file. I should be able to run your code both from the command line and from the REPL (when required) *as shown in the examples for the programs*. You don't need to use `argparse` for this assignment, but you can if you want. Don't forget to run `flake8` on all of the program files before submitting.
- For the programs that have command line interface *and* a function to be called from the REPL: When the program is run from the command line, *it should call the function*. In other words, *do not duplicate code* for use by the REPL and the command line.
- Review the 07a_files lecture to get some ideas. There is a separate document about using DictReader, please be sure to review it.
- If you spend more than an hour looking at a homework problem and have no idea what to do, **please email me.** Students in this class all have such varied levels of background that I don't always explain things in a way that is meaningful to *you*.

For those of you who are spending too much time Googling for answers, **do not use Pandas (or any other third party package).** It is not necessary for any of the homework programs, and **you will not receive full credit if you use Pandas.**

Program 1: file_stats. (15 pts)

Complete the program `file_stats.py` that, when run on the command line, accepts one text file name as an argument and outputs the number of characters (e.g., size of the file), words, lines, and the length (in characters) of the longest line in the file.

NOTE. This is the only one of the homework problems that prints.

Please note that this should work on **any** *text file, csv file or any other file that is text-based, including a* `.py` *file*. For the number of words, you want to separate words based on **default** "whitespace"; *string methods are your friends here.*

NOTE: If you are looping through the file counting things yourself, you're going it wrong! In particular, see slides 14 and 15 from the 07a_Files lecture. You will find useful things there.

The program file should have a **function** `stats` that takes the filename as an input and returns the statistics as a **tuple** (in the order shown), so it can be used in the REPL like this:

```
>>> from file_stats import stats
>>> chars, words, lines, longest = stats('beautiful_soup_by_Lewis_Carroll.txt')
>>> print(f"characters: {chars}, words: {words}, lines: {lines}, longest: {longest}")
characters: 553, words: 81, lines: 21, longest: 38
```

It should work like this from the command line:

```
$ python file_stats.py beautiful_soup_by_Lewis_Carroll.txt
Characters: 553
Words: 81
Lines: 21
Longest line: 38
$ python file_stats.py books.csv
Characters: 557
Words: 45
Lines: 6
Longest line: 120
```

The files `beautiful_soup_by_Lewis_Carroll.txt` and `books.csv` are provided in the files for your testing.

Program 2: tabs_to_commas. (15 pts)

Complete the program `tabs_to_commas.py` that when run on the command line, takes the name of a CSV file (or TSV file) that uses *tabs as delimiters* and converts it to one using *commas as delimiters* with the filename modified with `_commas` appended and the file extension changed to `csv`. This means that if a data field in the tab-delimited input file has tab characters in it, then the same data field should *still* have tab characters in the output file. So if the tab-delimited file you read is called `my_info.tsv`, the new file should be called `my_info_commas.csv`. Note: The file names are strings, so again, string methods are your friends.

Two files are included to test this that use the tab delimiter: `colors.csv` and `people.tsv`. Both files use the tab character as a delimiter.

I have included a file, `people_result.csv`, which contains the correct result from running the following command, which you can compare with the `people_commas.csv` to ensure your program is working.

Hint: It's *specifically* for csv-type files, so use the `csv` module. Do not try to substitute characters yourself!

```
$ python tabs_to_commas.py people.tsv
```

Results should be in the new file `people_commas.csv`. The file `people_commas.csv` should be the same as `people_result.csv`.

This program should have a **function** `tab_to_comma` that has two arguments: `in_file`, the name of the csv/tsv file to read, and `out_file`, the name of the csv file to create, so it can be used from the REPL or another Python module. The function arguments are in that order, namely (in_file, out_file). **The function should not modify the filenames given.**

```
>>> from tabs_to_commas import tab_to_comma
>>> tab_to_comma('people.tsv', 'people.csv')
```

Results in a new file `people.csv`.

Similarly, you can use the keyword arguments:

```
>>> from tabs_to_commas import tab_to_comma
>>> tab_to_comma(out_file='colors_converted.csv', in_file='colors.csv')
```

Will result in the creation of a new file `colors_converted.csv`. The new `colors_converted.csv` file should look like this in a code editor:

```
purple,0.15
indigo,0.25
crimson red,0.30
sky blue,0.05
spring green,0.25
```

Program 3: re_order. (20 pts)

Complete the program `re_order.py` that when run on the command line, takes 2 file names; an input csv file and output file name. It opens the input CSV file, reads it and writes out the data with the first and second columns switched.

Please note that this program/function must work for **any** *input csv file that has at least 2 columns, not just what is provided for your testing purposes.* For example, it should work on the `people_result.csv` file. You need not worry about checking for invalid files. For example, it should also work with a csv file that has **no header row,** such as the colors_commas.csv file created with the tabs_to_commas.py program. This is because we want to re-order the header (if there is one), so we don't care if there is a header.

You may assume that the input file will use the default delimiter (comma) and will always have at least 2 columns.

Hint: It's *specifically* for csv files, so use the `csv` module. Also, it might help to review how slicing works. DictReader/DictWriter is not useful here, because re_order should work whether or not there is a header file.

A file `books.csv` is provided to use as a test file.

```
$ python re_order.py books.csv books_author.csv
```

Results in the creation of a new file `books_author.csv` that contains the same data as in books.csv, with the first two columns (title and author) switched.

This program should have a **function** `re_order` that has two arguments: `in_file`, the name of the csv file to read, and `out_file`, the name of the csv file to create, so it can be used from the REPL or another Python module. **The function should not modify the filenames given.**

```
>>> from re_order import re_order
>>> re_order(in_file='books.csv', out_file='books_author.csv')
```

Will also result in the creation of a new file `books_author.csv`. Both versions of `books_author.csv` should be the same and match the file `books_author_result.csv`.

Program 4: re_sort. (20 pts)

Complete the program `re_sort.py` that when run on the command line, takes a filename of a CSV file (with header) as input, then reads the CSV file, sorts the file by the values of the **second** column, and writes the data to a new CSV file with `_sort` appended to the file name. So if the input filename is `books.csv`, the output file should be `books_sort.csv`. Assume the CSV file has 1 single header row.

Think about what might be a useful coding thing to use for a CSV file when we know there is a header row in the file. The output file should maintain the header row. You may want to review the Python HOWTO for using the `sorted` builtin function.

Hint: It's *specifically* for csv files, so use the `csv` module. Consider using DictReader/DictWriter, though it is not required. See the DictReader document included with this week's lesson.

Please note that this program/function must work for **any** *input csv file that has a header row and at least 2 columns, not just the one we use for an example.* You need not worry about checking for invalid files.

```
$ python re_sort.py books.csv
```

Will result in a file named `books_sort.csv`.

This program should have a **function** `re_sort` that has two arguments: `in_file`, the name of the csv file to read, and `out_file`, the name of the csv file to create, so it can be used from the REPL or another Python module. **The function should not modify the filenames given.**

```
>>> from re_sort import re_sort
>>> re_sort(in_file='books.csv', out_file='books_sorted.csv')
```

Will result in a file named `books_sorted.csv`. The files `books_sort.csv` and `books_sorted.csv` should match the file `books_sort_result.csv`.

Program 5: country_convert. (30 pts)

This is a different kind of program. It is written specifically for one file, and only needs to run from the command line. Complete the program `country_convert.py` that opens the included `countryInfo.csv` file and extracts the country name, capital city and population from each row, then writes a new file named `country_simple_info.csv` with country, capital and population in each row, with the rows sorted by population size, largest first. Note the `countryInfo.csv` file is **tab-delimited**, but we want `country_simple_info.csv` to use the *default* comma delimiter.

Note this program does not take input/output file names, as it is specific to the one file. It only needs to run on the command line like this:

```
$ python country_convert.py
```

Which will result in a file named `country_simple_info.csv` whose first 4 lines are:

```
country,capital,population
China,Beijing,1330044000
India,New Delhi,1173108018
United States,Washington,310232863
[...]
```

Hints: Don’t try to do everything all together. There are several steps involved here, so break the problem down into steps that will have intermediate data before getting the final data. Built-in functions are your friends. Note column headings. You don't need intermediate files.

I suggest using `DictReader` and `DictWriter` for this exercise. It's OK to read the whole file in at one time. See the DictReader document included with this week's lesson. Each element in the resulting list will be a *dictionary* of row data, containing the column name & cell value as the key, value pairs.

Then make a new list such that each element (namely, each row) consists of a dictionary that only contains the values (columns) from the initial list that we want.

Now, you can make a new list of row data, in order, by using the built-in function `sorted()`. Look up the inputs to `sorted()`, since the values of the dictionary are only strings, not numbers. You may want to define a helper function for the key function in `sorted`, or ou can use a lambda function.

Then write out the sorted list data with `DictWriter`. Don't forget that the column for the country name is different in the output file.

If you have a Windows computer: Be sure to use `newline=''` in the file open statements. Most importantly, don't print all the country or capital names in debug print statements (maybe just do the first few lines if you really think it's necessary), because, amazingly enough, the Windows Command window does NOT support display of UTF-8 characters, even in Windows 10, so your print statements will crash the program. This totally blows my mind. It might be OK in PowerShell.

My email is dianechen.ucsdext@gmail.com. Please do not hesitate to email me if you have questions.