Final Project Assignment

Please Note

This Final Project assignment involves the analysis of a population data set published by the Wisconsin Department of Health Services (WDHS). This data set divides the population into two categories: Male and Female. I can only assume that this categorization scheme is useful for the WDHS and its clients. For us, it represents a readily available data set that is easily understood. Those are my motivations for having chosen it.

Nevertheless, I am sure that many of you are aware that not everyone in our world identifies as either Male or Female. Rather, a significant number of people have a more nuanced gender identity. I want to assure you that I have the greatest respect for the gender identity of all of my students, my coworkers, and the general public. I encourage everyone to share in that point of view.

Assignment Overview

In this Final Project Assignment, you will be expected to create and use a series of related Python programs to analyze a sample population data set. The assignment is divided into 4 exercises. These exercises build upon each other, with results from one exercise being used in the next exercise. The goal is to provide an experience for you that is reasonably similar to an assignment that you might be given in the workplace.

The data set that you will be analyzing is population data for Milwaukee County from July 2014. Please note that you will NOT need to download the data set from WDHS. I have provided the data as one of the starter files for this assignment. This data set is one of many provided by the Wisconsin Department of Health Services (WDHS). Similar data sets are available for other Wisconsin counties and for other time periods. While you will only be analyzing one data set in this Final Project, you can imagine that the code that you create will be eventually used by others to analyze data sets for other counties and other time periods for which data sets are available from WDHS.

You will be working more independently on this assignment than you have on the weekly coding assignments from earlier in the semester. I will not be providing a tutorial video to prepare you for each exercise in this assignment. Instead, I will be providing an overview tutorial video for the entire Final Project that helps you understand the assignment more completely and how the work flows through the 4 exercises. More important, the instructions for each assignment will include more direction than the instructions from the weekly coding assignments. This direction is meant to resemble specifications that a supervisor might provide to a junior programmer in the workplace.

Despite this lesser level of tutorial direction on this assignment, you can expect the same level of help and support from me. I am willing to discuss problems and strategies during the remaining Online Lab Sessions. Also, I will continue to support your individual inquiries via the Service Desk for this course. I want you to do as well as possible on this assignment. So, if you are stuck on any part of the Final Project, please seek my help as soon as possible.

Also, you should feel free to discuss strategies and problems with the Final Project with your classmates. Conferring with each other is fair game. Feel free to show someone your broken code and ask for advice. By contrast, giving your working code to other students is not allowed. We want to offer help, not the code itself.

General Expectations for Work Submitted

When completing your work on the Final Project, you are expected to follow all of good practices that we have covered during our course. Here is a summary of those good practices:

- Include a single-line comment with name of program file.
- Include a single-line comment that describes the intent of the program.
- Place your highest-level code in a function named main.
- Your code should be factored such that there is a function in your program for each part of the problem.
- Each function should contain code relating to the same thing – it should have *high cohesion*.
- Functions should know as little as possible about the workings of other functions – they should have *low coupling*.
- If the Python file that you are creating is a regular executable program, include a final line of code in the program that calls the *main()* function.
- Follow all PEP-8 Python coding style guidelines enforced by the PyCharm Editor. For example, place two blank lines between the code making up a function (or class) and the code that surrounds that function (or class).
- Output printed by the program (both prompts and results) should be polite and descriptive.
- Choose names for your variables that are properly descriptive.
- Choose names for your functions that are properly descriptive.
- Choose names for your classes that are properly descriptive
- Follow PEP-8 Python coding style guidelines for forming names of variables, functions, and classes.
- Close all files before the conclusion of the program.
- Model your solution after the code that I demonstrate in the tutorial videos.
- Remember to test your program thoroughly before submitting your work.
- Your code must pass all relevant test cases. Make sure that it passes tests at the boundaries created by *if, else*, and *elif* conditions in your program (boundary value tests).
- If the python file that you are creating is a module that acts as a container for a reusable Python class, then:
    - Place your unit testing code for the class in the *main()* function.
    - Include statements at the end of your module file that cause the *main()* function to be called only when the module is run directly.
    - Make sure that the code in *main()* is not called when the module is imported into another program.

- When coding a Python class:
    - Make sure that you name the instance variables, the methods, and the class itself **exactly** as specified in the instructions. Classes are often shared with other developers. It is important that everyone involved in sharing the class knows what names to expect for instance variables, methods, and the class itself.
    - Make sure that all client code can access instance variables using Pythonic field access (*instance.fieldname).*
    - When typical getter/setter features are needed for an instance variable, implement Pythonic getter/setter features using the *@property* decorators. DO NOT create Non-Pythonic getter/setter methods with names like *get_fieldname()* and *set_fieldname()*.
    - Never store values as instance variables that may be derived from other instance variables. Instead, provide methods in the class with names like *calculate_derived_value().*
    - Always provide an *__init__()* constructor.
    - Always provide a *__str__()* method.
    - Always provide a *__repr__()* method.

Starter Files

I have provided starter files for this project in the following ZIP file:

- starter_files_for_python_course_final_project.zip

The ZIP file contains the following data files:

- raw_data.txt
- cleaned_data.txt

Please note that these two files have the same contents. You will be editing the *cleaned_data.txt* file, making cleaning corrections to the data. When fully cleaned, this is the file that you will use as input data during subsequent exercise steps. The *raw_data.txt* file is only provided as a restarting point in case changes to the cleaned data get so confused that you want to start over from the beginning.

Exercise 1

Create a Python program named *detect_row_level_data_entry_errors*. When complete, you will run this program to produce a diagnostic report that shows data entry errors that cause row totals in the data to be out of balance. You will be expected to edit the *cleaned_data.txt* input file and make corrections until running this program shows that all row-level errors have been resolved.

I must confess that I created these data entry errors on purpose by corrupting the data that I downloaded from WDHS. Data cleaning is an important part of data science and I wanted you to have the experience of creating programs to support this task. I have provided a PDF copy of the original data set in Appendix A of these directions. You should feel free to refer to these data while you are finding and correcting data entry errors. An enterprising student might conclude that one could correct all of the data simply by proofreading. Nevertheless, you are expected to create this program that automatically exposes data entry errors. While it might be feasible to correct this one data set by proofreading, it would not be feasible to take the same approach to correcting data entry errors for a large volume of these data sets.

A first run of this program should provide the following console session output:

```
Please enter the input filename: cleaned_data.txt

           Row-Level Data Entry Errors

Age Group      Males   Females     Total     Error
0-14          97,680    93,991   191,671         0
15-19         32,800    32,479    65,319       -40
20-24         38,953    41,206    80,159         0
25-29         36,775    38,205    74,980         0
30-34         37,072    39,197    76,269         0
35-39         30,337    31,644    61,801       180
40-44         28,176    29,271    57,447         0
45-54         57,519    60,283   117,802         0
55-64         52,893    57,669   110,562         0
65-74         28,577    34,212    61,999       790
75-84         13,843    20,222    34,065         0
85+            5,775    12,843    18,618         0
Total        460,340   491,642   951,982         0
```

Your next activity will be to solve the errors reported at the row level. A positive error indicates that the sum of the Males value and the Females value is greater than the stated Total value. A negative error indicates that the sum is less than the stated Total value. Refer to the printed copy of the dataset available in Appendix A to identify and make the corrections needed. To accomplish this, you will be using at text editor to correct the values in `cleaned_data.txt`.

Continue running this program and making corrections in the cleaned_data.txt file until all row-level data entry errors have been corrected and the output from the console session appears as follows:

```
Please enter the input filename: cleaned_data.txt

           Row-Level Data Entry Errors

Age Group      Males   Females     Total     Error
0-14          97,680    93,991   191,671         0
15-19         32,840    32,479    65,319         0
20-24         38,953    41,206    80,159         0
25-29         36,775    38,205    74,980         0
30-34         37,072    39,197    76,269         0
35-39         30,337    31,464    61,801         0
40-44         28,176    29,271    57,447         0
45-54         57,519    60,283   117,802         0
55-64         52,893    57,669   110,562         0
65-74         28,577    34,212    62,789         0
75-84         13,743    20,822    34,565         0
85+            5,775    12,843    18,618         0
Total        460,340   491,642   951,982         0
```

Please note: This program does NOT need to store data in custom objects based upon a custom Python class. The input file will only need to be processed once. The file can be processed in its current order. It does not need to be sorted.

The following is pseudocode that you may use to help design your program:

```
prompt for input filename
open input file using encoding utf8
print report heading

for line in input file:
      split line into individual strings
      convert stings to ints as appropriate
      calculate row total
      calculate row error
      print line for a row (including error)

close file
```

When formatting the report lines, remember the following hints:

- Report lines are 50 characters wide.
- Each column is 10 characters wide.
- The report title is centered within the 50-character line.
- A format string that will be useful on the report title is: `{0:^50}`
- A format string that will be useful on the report detail line is: `{1: > 10,}`

Exercise 2

Create a Python program named *detect_column_level_data_entry_errors*. When complete, you will run this program to produce a diagnostic report that shows data entry errors that cause column totals in the data to be out of balance. You will be expected to edit the *cleaned_data.txt* input file and make corrections until running this program shows that all column-level errors have been resolved.

Please remember the advice that I gave above regarding the need to write a program and use it to correct the column-level data errors. I realize that it can be done by proofreading alone. Nevertheless, your job is to create a program that automates error detection.

Provided that you have already corrected the row-level data entry errors during Exercise 1, a first run of this program should provide the following console session output:

```
Please enter the input filename: cleaned_data.txt

     Column-Level Data Entry Errors

Age Group      Males   Females     Total
0-14          97,680    93,991   191,671
15-19         32,840    32,479    65,319
20-24         38,953    41,206    80,159
25-29         36,775    38,205    74,980
30-34         37,072    39,197    76,269
35-39         30,337    31,464    61,801
40-44         28,176    29,271    57,447
45-54         57,519    60,283   117,802
55-64         52,893    57,669   110,562
65-74         28,577    34,212    62,789
75-84         13,843    20,222    34,065
85+            5,775    12,843    18,618
Total        460,340   491,642   951,982
Error           -100       600       500
```

Your next activity will be to solve the errors reported at the column level. Error values are computed for each column. A positive error indicates that the sum of the values for each age group category is greater than the stated column total value. A negative error indicates that the sum of the values for each age group category is less than the stated column total value. Refer to the printed copy of the dataset available in Appendix A to identify and make the corrections needed. To accomplish this, you will be using at text editor to correct the values in `cleaned_data.txt`.

Continue running this program and making corrections in the cleaned_data.txt file until all column-level data entry errors have been corrected and the output from the console session appears as follows:

```
Please enter the input filename: cleaned_data.txt

     Column-Level Data Entry Errors

Age Group      Males   Females     Total
0-14          97,680    93,991   191,671
15-19         32,840    32,479    65,319
20-24         38,953    41,206    80,159
25-29         36,775    38,205    74,980
30-34         37,072    39,197    76,269
35-39         30,337    31,464    61,801
40-44         28,176    29,271    57,447
45-54         57,519    60,283   117,802
55-64         52,893    57,669   110,562
65-74         28,577    34,212    62,789
75-84         13,743    20,822    34,565
85+            5,775    12,843    18,618
Total        460,340   491,642   951,982
Error              0         0         0
```

Please note: This program does NOT need to store data in custom objects based upon a custom Python class. The input file will only need to be processed once. The file can be processed in its current order. It does not need to be sorted.

The following is pseudocode that you may use to help design your program:

```
prompt for input filename
open input file using encoding utf8
initialize accumulators for males, females, total

print report heading
for line in input file:
     split line into individual strings
     convert stings to ints as appropriate
     print line for a row
     if age group category name shows this is NOT the total line:
          add values from this line to accumulators
     else:
          compute the column error values
print the column error line
close file
```

When formatting the report lines, remember the following hints:

- Report lines are 40 characters wide.
- Each column is 10 characters wide.
- The report title is centered within the 40-character line.
- A format string that will be useful on the report title is: `{0:^40}`
- A format string that will be useful on the report detail line is: `{1: > 10,}`

Please continue to Exercise 3 (below)

Exercise 3

Create a Python module file named *my_population_groups.py.* This module will hold the *PopulationGroup* class and related test code. In this exercise, you will be creating the *PopulationGroup* class and conducting a full unit test. In Exercise 4, you will be importing this module and using the *PopulationGroup* class to create a series of analysis reports.

The requirements for the *PopulationGroup* class include the following:

- An instance variable *category* that is expected to hold a string.
- An instance variable *male_count* this is expected to hold an int.
- An instance variable *female_count* this expected to hold an int.
- A method *calculate_total_count* that is expected to return an int.
- A @property-based getter/setter pair for the *category* instance variable. *category* may not be set to the empty string.
- A @property-based getter/setter pair for the *male_count* instance variable. *male_count* may not be set to a value less than zero.
- A @property-based getter/setter pair for the *female_count* instance variable. *female_count* may not be set to a value less than zero.
- An __*init*__ constructor that allows all instance variables to be set.
- A __str__ method that returns a proper string representation of the *PopulationGroup* object.
- A __repr__ method that returns a proper string representation of the *PopulationGroup* object.

Please note: you must use the exact names listed above for instance variables, the methods, and the class itself. Using different names will result in points being lost when this exercise is graded.

When coding and testing this module, I recommend that you refer to the separate document *Cheat Sheet: Create Custom Python Class*. A link to this document has been provided in the weekly schedule.

The module should also contain a *main()* function the contains unit test code for the PopulationGroup class. This unit test code should reflect the standards and good practices for unit testing of classes that have been demonstrated in our course.

When this program is run directly (rather than having been imported), the console session should contain the unit testing output and should look like this:

```
Unit testing output follows...

Test 1:  Test Constructor
Passed

Test 2:  Attempt to set category attribute to empty string
Passed

Test 3:  Attempt to set male_count attribute to negative value
Passed

Test 4:  Attempt to set female_count attribute to negative value
Passed

Test 5:  Test calculate_total_count() method
Passed

T Test 6:  Test __str__ Method
Passed

Test 7:  Test __repr__ Method
Passed
```

Please continue to Exercise 4 (below)

Exercise 4

Create a Python program named *create_data_analysis_reports*. When complete, you will run this program to produce a series of eight data analysis reports that will help you and your clients understand the data set. These reports include:

- Counts by Age Group
- Percentages by Age Group
- Counts by Descending Total Count
- Percentages by Descending Total Count
- Counts by Descending Female Count
- Percentages by Descending Female Count
- Counts by Descending Male Count
- Percentages by Descending Male Count

These eight reports are expected to print in the order shown. Interleaving the Counts-based reports with the Percentages-based reports means that the data will need to be sorted half as many times. Since this is a significant savings in processing, we will want to take advantage of it.

When this program is run, the following console session output should be generated:

```
Please enter the input filename: cleaned_data.txt


          Counts by Age Group

Age Group      Males   Females     Total
0-14          97,680    93,991   191,671
15-19         32,840    32,479    65,319
20-24         38,953    41,206    80,159
25-29         36,775    38,205    74,980
30-34         37,072    39,197    76,269
35-39         30,337    31,464    61,801
40-44         28,176    29,271    57,447
45-54         57,519    60,283   117,802
55-64         52,893    57,669   110,562
65-74         28,577    34,212    62,789
75-84         13,743    20,822    34,565
85+            5,775    12,843    18,618
Total        460,340   491,642   951,982
```

Percentages by Age Group

Age Group	Males	Females	Total
0-14	21.22%	19.12%	20.13%
15-19	7.13%	6.61%	6.86%
20-24	8.46%	8.38%	8.42%
25-29	7.99%	7.77%	7.88%
30-34	8.05%	7.97%	8.01%
35-39	6.59%	6.40%	6.49%
40-44	6.12%	5.95%	6.03%
45-54	12.49%	12.26%	12.37%
55-64	11.49%	11.73%	11.61%
65-74	6.21%	6.96%	6.60%
75-84	2.99%	4.24%	3.63%
85+	1.25%	2.61%	1.96%
Total	100.00%	100.00%	100.00%

Counts by Descending Total Count

Age Group	Males	Females	Total
0-14	97,680	93,991	191,671
45-54	57,519	60,283	117,802
55-64	52,893	57,669	110,562
20-24	38,953	41,206	80,159
30-34	37,072	39,197	76,269
25-29	36,775	38,205	74,980
15-19	32,840	32,479	65,319
65-74	28,577	34,212	62,789
35-39	30,337	31,464	61,801
40-44	28,176	29,271	57,447
75-84	13,743	20,822	34,565
85+	5,775	12,843	18,618
Total	460,340	491,642	951,982

Percentages by Descending Total Count

Age Group	Males	Females	Total
0-14	21.22%	19.12%	20.13%
45-54	12.49%	12.26%	12.37%
55-64	11.49%	11.73%	11.61%
20-24	8.46%	8.38%	8.42%
30-34	8.05%	7.97%	8.01%
25-29	7.99%	7.77%	7.88%
15-19	7.13%	6.61%	6.86%
65-74	6.21%	6.96%	6.60%
35-39	6.59%	6.40%	6.49%
40-44	6.12%	5.95%	6.03%
75-84	2.99%	4.24%	3.63%
85+	1.25%	2.61%	1.96%
Total	100.00%	100.00%	100.00%

Counts by Descending Female Count

Age Group	Males	Females	Total
0-14	97,680	93,991	191,671
45-54	57,519	60,283	117,802
55-64	52,893	57,669	110,562
20-24	38,953	41,206	80,159
30-34	37,072	39,197	76,269
25-29	36,775	38,205	74,980
65-74	28,577	34,212	62,789
15-19	32,840	32,479	65,319
35-39	30,337	31,464	61,801
40-44	28,176	29,271	57,447
75-84	13,743	20,822	34,565
85+	5,775	12,843	18,618
Total	460,340	491,642	951,982

Percentages by Descending Female Count

Age Group	Males	Females	Total
0-14	21.22%	19.12%	20.13%
45-54	12.49%	12.26%	12.37%
55-64	11.49%	11.73%	11.61%
20-24	8.46%	8.38%	8.42%
30-34	8.05%	7.97%	8.01%
25-29	7.99%	7.77%	7.88%
65-74	6.21%	6.96%	6.60%
15-19	7.13%	6.61%	6.86%
35-39	6.59%	6.40%	6.49%
40-44	6.12%	5.95%	6.03%
75-84	2.99%	4.24%	3.63%
85+	1.25%	2.61%	1.96%
Total	100.00%	100.00%	100.00%

Counts by Descending Male Count

Age Group	Males	Females	Total
0-14	97,680	93,991	191,671
45-54	57,519	60,283	117,802
55-64	52,893	57,669	110,562
20-24	38,953	41,206	80,159
30-34	37,072	39,197	76,269
25-29	36,775	38,205	74,980
15-19	32,840	32,479	65,319
35-39	30,337	31,464	61,801
65-74	28,577	34,212	62,789
40-44	28,176	29,271	57,447
75-84	13,743	20,822	34,565
85+	5,775	12,843	18,618
Total	460,340	491,642	951,982

```
  Percentages by Descending Male Count

Age Group      Males   Females     Total
0-14          21.22%    19.12%    20.13%
45-54         12.49%    12.26%    12.37%
55-64         11.49%    11.73%    11.61%
20-24          8.46%     8.38%     8.42%
30-34          8.05%     7.97%     8.01%
25-29          7.99%     7.77%     7.88%
15-19          7.13%     6.61%     6.86%
35-39          6.59%     6.40%     6.49%
65-74          6.21%     6.96%     6.60%
40-44          6.12%     5.95%     6.03%
75-84          2.99%     4.24%     3.63%
85+            1.25%     2.61%     1.96%
Total        100.00%   100.00%   100.00%
```

The recommended way to produce these reports is to create two reusable methods: one method to create a Count-based report, and the other method to create a Percentage-based report. When testing your code, this probably means that you will test all 4 of the Count-based reports before you start coding and testing for the 4 Percentage-based reports. Just remember that before you are finished, you need to be sure that the reports are printing in the proper interleaved order.

The following is pseudocode that you may use to help design your *main* function:

```
do build_population_group_list
do calculate_column_totals

sort population groups by category
do create_count_based_report
do create_percentage_based_report

sort population groups by total_count descending
do create_count_based_report
do create_percentage_based_report

sort population groups by female_count descending
do create_count_based_report
do create_percentage_based_report

sort population groups by male_count descending
do create_count_based_report
do create_percentage_based_report
```

The following is pseudocode that you may use to help design your *build_population_group_list* function:

```
prompt for infile
open infile with encoding utf8
initalize population_groups_list

for line in infile:
     split line into strings
     convert male_count and female_count to ints
     if this line is NOT the total line:
          construct a new Population Group instance
          append instance to population_groups list
close infile
return population_groups_list
```

The following is pseudocode that you may use to help design your *calculate_column_totals* function:

```
Receive population_groups_list as parameter
initialize male_total, female_total, overall_total

for group in population_groups_list:
     accumulate male_total, female_total, overall_total

return male_total, female_total, overall_total
```

The following is pseudocode that you may use to help design your *create_count_based_report* function:

```
receive population_groups_list, male_total, female_total,
     overall_total, title as parameters
print blank lines
print title
print column headings

for group in population_groups_list:
     print a report line using values from PopulationGroup instance

print column total line using male_total, female_total, overall_total
```

The following is pseudocode that you may use to help design your *create_percentage_based_report* function:

```
receive population_groups_list, male_total, female_total,
      overall_total, title as parameters
print blank lines
print title
print column headings

for group in population_groups_list:
      use group instance to get male_count, female_count,
            total_count
      calculate percentages based upon male_total,
            female_total, overall_total
      print a report line using percentages

print column total line where all values are 100%
```

When formatting the report lines, remember the following hints:

- Report lines are 40 characters wide.
- Each column is 10 characters wide.
- The report title is centered within the 40-character line.
- A format string that will be useful on the report title is:
  - `{0:^40}`
- A format string that will be useful on a Counts-based report detail line is:
  - `{1: > 10,}`
- A format string that will be useful on a Percent-based report detail line is:
  - `{1: > 10.2%}`

Tools
Use PyCharm to create and test all Python programs.

Submission Method
Follow the process that I demonstrated in the tutorial video on submitting your work. This involves:

- Locating the properly named directory associated with your project in the file system.
- Compressing that directory into a single .ZIP file using a utility program.
- Submitting the properly named zip file to the submission activity for this assignment.

File and Directory Naming
Please name your Python program files as instructed in each exercise. Please use the following naming scheme for naming your PyCharm project:

```
surname_givenname_final_project
```

If this were my own project, I would name my PyCharm project as follows:

```
trainor_kevin_final_project
```

Use a zip utility to create one zip file that contain the PyCharm project directory. The zip file should be named according to the following scheme:

```
surname_givenname_final_project.zip
```

If this were my own project, I would name the zip file as follows:

```
trainor_kevin_final_project.zip
```

Due By
Please submit this assignment by the date and time shown in the Weekly Schedule.

Last Revised
2021-03-25

Appendix A

Milwaukee County: July 1, 2014 Population

Age Group	Males	Females	Total	Percent Change from 2010
0-14	97,680	93,991	191,671	-3%
15-19	32,840	32,479	65,319	-7%
20-24	38,953	41,206	80,159	3%
25-29	36,775	38,205	74,980	-4%
30-34	37,072	39,197	76,269	12%
35-39	30,337	31,464	61,801	3%
40-44	28,176	29,271	57,447	-3%
45-54	57,519	60,283	117,802	-7%
55-64	52,893	57,669	110,562	9%
65-74	28,577	34,212	62,789	21%
75-84	13,743	20,822	34,565	-10%
85+	5,775	12,843	18,618	-2%
Total	460,340	491,642	951,982	0%

Age Group	Males	Females	Total	Percent Change from 2010
0-17	117,230	113,366	230,596	-2%
18-44	184,603	192,447	377,050	1%
45-64	110,412	117,952	228,364	0%
65+	48,095	67,877	115,972	6%
Total	460,340	491,642	951,982	0%

Source: Office of Health Informatics, Division of Public Health, Wisconsin Department of Health Services