CISC 333: Final Project

## Contents

[1 Introduction 1](#_Toc36733927)

[2 Program Specifications 3](#_Toc36733928)

[3 Design by Contract 4](#_Toc36733929)

[4 Analysis of Code 4](#_Toc36733930)

[5 Time Estimation 4](#_Toc36733931)

[6 Black-Box Testing 5](#_Toc36733932)

[7 White-Box Testing 5](#_Toc36733933)

# Introduction

Your software development company has been brought in on a contract with a local electric utility. This utility has recently had an influx of customers who are installing solar panels and are receiving exponentially more questions about billing. (“How does this work?” “What will my bill be?” “How likely am I to generate more energy than I use?”) In an attempt to help, Walt from accounts receivable slammed together a simple, first draft of an application to answer some of these questions. While the utility does not expect the current application to be used and expect it to be replaced, they would like the first step of the process to be an analysis of the current code to see what, if anything, can be polished and repurposed in some way.

Your job is to provide this analysis. They have provided you with the original source code, a sample data file, and a design specification (see Section 2 of this document). As near as they can tell, the program currently works.

Your answers may be provided in this booklet or in a separate file. If you provide your answers in a separate file, your solutions must be clearly labeled. You do not need to provide code for any of these solutions, including generation of test cases although code is would be an acceptable submission. However, if you elect to provide a description for test cases, be sure that the description is robust and complete.

**Any coding should be done and submitted in an external file instead of in electric\_usage.py. You do not need to submit and I will not be examining any submitted copies of electric\_usage.py.**

**Reminder: DO NOT SUBMIT YOUR SOLUTIONS IN A ZIP FILE. ZIPPED SUBMISSIONS WILL NOT BE GRADED.**

# Program Specifications

This program was designed to handle a specific file format which can be generated by a customer. When a customer calls with questions, the customer service agent will generate the file and then run the program. The file name CAN be specified at command line (example: python electric\_usage.py other\_file.txt) or as a parameter to the main function. However, if no file name is specified, then the program will look for a file called usage-data.txt by default.

This file is a multiuse file by different departments. However, the utility has been able to repurpose it for this application. The file is of a tab-delimited format with a header row of for column labels (Year followed by the twelve months). Each row starts with a label for the year that the data was generated from, followed by the data for each month of that year. If there is no data for a given month, then the data for the month will just be blank. This most frequently happens when a customer starts service or requests information in a month other than January.

The data for any given month has four elements, separated by semicolons. The first element represents the nature of the bill. These are typically DEBIT which means that the customer used more electricity than the generated or CREDIT which means the customer generated more electricity than they used. There are other classifications that COULD be seen in this file for special situations (DEFER, SUBSIDY, etc.) that have no bearing on the current application and, thus, can be ignored.

The second element of the monthly data represents the amount that the customer will be debited or credited on that month’s bill. This number should always be positive with the first element determining if the value is treated positive (DEBIT) or negative (CREDIT). The third value represents the amount of electricity delivered to the customer that month. The fourth value represents the amount of electricity received from the customer that month.

Once the file is read and analyzed, the customer service agent can ask for one of three reports. Each report is specified by a letter and that letter in either case should work. The first option (M) displays the number of months that the data represents. The second option (A) the average of the three data points in the file where credit amounts are represented as negative number. There is also a fourth data point that represents the difference between the actual total of the net usage data in the file and the calculated net usage (delivered – received, summed over all months). The third option (C) will ask the agent for a price per kilowatt hour (kWh) and calculate the customers average, minimum, and maximum monthly costs, based on the price provided and historical data. The final option (Q) quits the program. No other inputs should be accepted and should return the agent to the menu.

Finally, as our agents are not particularly tech savvy, we want to make sure that there are as few crashes or instances of “technobabble” displayed to the user as is reasonably possible. All anticipatable errors should be handled within the program and the program should return to the command prompt only as a truly rare and unexpected occasion.

This question is asking for the design and implementation of a Python program that handles a specific file format used by customers. The program should be able to read and analyze the data from the file, and provide various reports based on user input.

Here's a breakdown of the requirements:

1. The program should be able to handle a file in a tab-delimited format with a header row specifying column labels (Year followed by the twelve months).
2. Each row in the file represents data for a specific year, with data for each month of that year. If there is no data for a month, it will be represented as blank.
3. The monthly data consists of four elements separated by semicolons: nature of the bill, amount (positive for debit, negative for credit), amount of electricity delivered, and amount of electricity received.
4. The program should provide three types of reports:
   * M: Display the number of months that the data represents.
   * A: Calculate and display the average of the three data points in the file where credit amounts are represented as negative numbers.
   * C: Calculate and display average, minimum, and maximum monthly costs based on the price per kilowatt hour provided by the user and historical data.
   * Q: Quit the program.
5. The program should handle errors gracefully and avoid displaying technical jargon to non-tech-savvy users.

In summary, the task involves designing and implementing a Python program that can parse and analyze a specific file format, generate reports based on user input, and ensure a user-friendly experience for customer service agents.

Top of Form

Bottom of Form

# Design by Contract

By examining the file, it should be obvious that none of the functions are defined with an eye towards design by contract. There are a few blocks of assertions but you get the impression that these blocks are insufficient for supporting the actual needs of the functions.

Document the preconditions, postconditions, and invariants for the following functions: readFile, analyzeFile, printMenuAndGetInput, monthCount, analyzeData, and calculateExpenses. This documentation does not need to be in any particular format. However, it should be understandable by another developer and as comprehensive as possible. **(20 pts)**

**NOTE**: You **DO NOT** have to program corresponding assertion/if statements, simply document the functions.

**Answers in Question 3**.py

# Analysis of Code

Now that you have a better understanding of how the code works, we can engage in an analysis of the code itself. Analyze the code and note any particular strengths or weaknesses based on the principles that we learned in class this semester (don’t repeat yourself, cohesion, coupling, exception handling, resource management, scoping, naming, etc).

For any problems that you note with the code, provide an idea for how this can be revised. These revisions do not need to be examples of actual code. Instead, a robust description would be sufficient. (Example: The contents of the for loop in function X are repeated several places throughout the code, include functions Y and Z. Moving that code to its own function and then calling the function here would be preferable.) **(25 pts)**

**Answers in Question 4.py**

# Time Estimation

Next, we need to know about how fast our code will run because the only thing worse than bad code is slow, bad code. We want a run-time estimation in big-O notation for each of the following functions: readFile, analyzeFile, printMenuAndGetInput, monthCount, analyzeData, and calculateExpenses.

When you give the run-time complexity, you need to be able to justify it using techniques you learned previously or techniques you used in class. Be sure for each justification that you define any variables used. (In other words, don’t say O(n) without telling me what n represents.) Failure to define variables will result in a significant penalty to your score. **(10 pts)**

readFile:

The big O notation of readFile is O(n) where n is the number of characters in the file to be split up. This is because all the other statements in the function have a complexity of O(1) except for the nested for loop which has a notation of O(n) for the first loop and a notation of O(1) for the second loop. Multiplying all of the notations together, you get the notation of O(n)

analyzeFile:

The big O notation of analyzeFile is O(nm) where n is the number of rows in the tables and m is the number of columns in the tables. This is because in the function there is a nested loop which counts by the number of rows(n) and then the second loop counts by the number of columns(m). And because of all the other statements in this function having a complexity of O(1), this causes the big O notation of analyzeFile to be O(nm)

printMenuAndGetInput:

The big O notation of printMenuAndGetInput is O(n) where n is the length of ‘INPUTS’ because of the simple for loop that occurs in the function which causes the notation to be O(n).

monthCount:

The notation of monthCount is O(1) because its only printing out the variable ‘numberMonths’.

analyzeData:

The notation of analyzeData is O(1) because it is only printing out the value of a division of two variables which causes no time complexity.

calculateExpenses:

The big O notation of calculateExpenses is O(nm) where n is the number of rows and m is the number of columns in ‘netTable’. This is because there is a nested loop in the function which causes the first loop to have a notation of O(n) and the second loop to have a O(m). Multiplying these together, the notation becomes O(nm).

# Black-Box Testing

Now we can begin testing to insure correctness. First, we should engage in black-box testing, using techniques we learned in class, in order to ensure that the application works as defined. The tests must test the entire code base.

While you do not have to write code for this question, you have the functional code and could, if you wanted to, write the test code. This will give YOU a better idea of how you are doing and if you’re missing any behaviors or conditions. However, if you feel that you are not a strong coder, that is fine. You may write robust descriptions of each test case. Regardless of whether you submit code or a description, you must explain the purpose of the test case (i.e., what specific condition(s) you are testing with this case).

Keep in mind, you can engage in black-box testing on any level. I will expect you to do black-box testing both at the application and the functional level. **(25 pts)**

Test Case 1:

Input - A valid file with valid format

Expected Output – Program successfully reads and analyzes the file data

Purpose – Verify that the program can successfully read and analyze a file with the correct format and data

Test Case 2:

Input – File with incorrect data and/or formatting

Expected Output – An error message that is easy to understand is returned

Purpose- Verify that the program can handle invalid file inputs

Test Case 3:

Input – Do not input a file name

Expected Output – Program looks for a file with the name “usage-data.txt” to read and analyze the data

Purpose- Verify that the program handles the scenario where the user doesn’t input a file correctly

Test Case 4:

Input – Choose report option ‘M’

Expected Output – The program will return the correct number of months that were in the data file

Purpose- Verify that the program handles printing out the number of months correctly and that the number printed out is correct.

Test Case 5:

Input – Choose report option ‘A’

Expected Output – The program will calculate and display the average of the three data points where credit amounts are represented as negative numbers

Purpose- Verify that the program accurately calculates and displays the average of the three datapoints

Test Case 6:

Input – Choose report option ’C’ and provide a valid price

Expected Output – The program will calculate and display the average, minimum, and maximum monthly costs based on the provided price and historical data

Purpose- Verify that the program calculates and displays the values correctly

Test Case 7:

Input – Choose report option ‘Q’

Expected Output – The program will terminate without any errors

Purpose- Verify that the program will terminate without any issues

Test Case 8:

Input – Choose an invalid report option

Expected Output – The program will effectively ignore the input and go back to the menu

Purpose- Verify that the program handles invalid options correctly

Test Case 9:

Input – The input data file includes a blank row of data for a specific month

Expected Output – The program will skip over the blank data and not consider it in calculations

Purpose- Verify that the program handles this situation correctly and does not encounter either an error or include it in the calculations

# White-Box Testing

Finally, we can engage in white-box testing, using techniques we learned in class, in order to ensure that the application works as defined. The tests must much of the code base as possible.

While you do not have to write code for this question, you have the functional code and could, if you wanted to, write the test code. This will give YOU a better idea of how you are doing and if you’re missing any behaviors or conditions. However, if you feel that you are not a strong coder, that is fine. You may write robust descriptions of each test case. Regardless of whether you submit code or a description, you must explain the purpose of the test case (i.e., what specific condition(s) you are testing with this case).

Since white-box testing only makes sense for us at a functional level, I will only expect you to engage in white-box testing at the functional level. **(25 pts)**

**readFile Function Tests**

Test Case 1:

Input – Valid file with correct formatting

Expected Output – The global list variables are populated and filled with data

Purpose- Verify that the function works under optimal conditions

Test Case 2:

Input – Invalid file

Expected Output – returns an error message stating that the file could not open

Purpose- Verify that the except statement works when given an invalid file

Test Case 3:

Input – Valid file but the file is empty

Expected Output – Function runs without adding any values

Purpose- Verify that the function works at extreme values

Test Case 4:

Input – Valid file and DEBUG is True

Expected Output – The global list variables are populated and filled with data as well as the current month being printed

Purpose- Verify that the DEBUG block of code works correctly