### 



#### 

Module Code: CPUF001

Module Title: Software Foundation

**Assignment:** S1. Development Project

**Student name**

**Student id:**

#### 

#### *Table of Contents*

[*Table of Contents* 2](#_Toc196418608)

[Introduction 3](#_Toc196418609)

[Solution Design 3](#_Toc196418610)

[Flowchart 3](#_Toc196418611)

[Walkthrough 3](#_Toc196418612)

[Design 4](#_Toc196418613)

[Reflective Evaluation 6](#_Toc196418614)

[References 7](#_Toc196418615)

[Appendix 1: Source code 8](#_Toc196418616)

[Appendix 2: Script file 9](#_Toc196418617)

#### 

#### Introduction

For this development project (CPUF001), I implemented a Python based Data Processing application capable of reading numerical data from an input CSV file, performing two core calculations (sum and mean), and writing the results to a user specified output file. I interpreted the brief to emphasise command line argument handling, error management for file I/O and data parsing, and modular code structure. Throughout development I applied exception handling, incremental data processing for efficiency, and clear code documentation.

The complete source code and accompanying script are hosted in my GitHub repository:

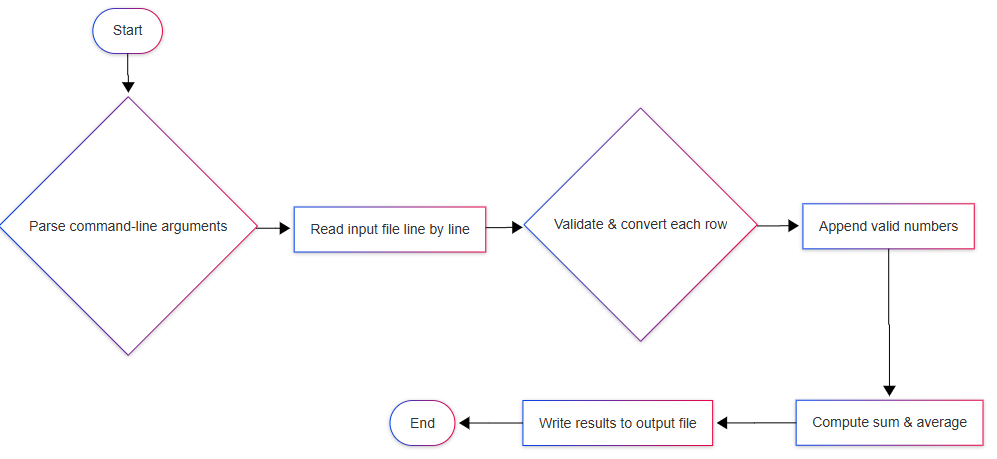
<https://github.com/avramos4/CPUF001---Software-Foundations-Assignment-1.git>

This document includes my solution design (with flowchart), technical code breakdown, and a reflective evaluation of lessons learned.

#### Solution Design

##### Flowchart

The following flowchart demonstrates the overall flow of the code:



##### Walkthrough

1. **Parse command-line arguments**

Use sys.argv to get input\_file and output\_file.

1. **Read input file**

Open CSV in read mode and skip the header.

1. **Validate & convert**

For each row, attempt float(row[1]); on failure, log an error.

1. **Collect data**

Append only successfully converted numbers into a list.

1. **Compute**

Use sum() for total and divide by len(data) for average (default 0 if empty).

1. **Write results**

Open the output\_file in write mode and dump two lines:

“Total Sum: …” and “Average: …”.

##### Design

The program has a straightforward and systematic order to accommodate the needs of the project in terms of effectiveness and efficiency of data processing. The systematic order can be subdivided into numerous well-determined steps, which are such that each of them makes the program carry out its operations with correctness and efficiency.

The program starts by taking the input file as a command-line argument. This is achieved by Python's sys.argv to take the file path that the user inputs when executing the script. Through this, the program can declare the file path of the input file and the output file that can be passed conveniently during runtime without coding in file paths into the program. After receiving the file path, the program then opens the input file in Python using the csv module. It is chosen because it provides an easy means of reading CSV files, which are usually used during data processing activities. The csv.reader function is used to open the file, and the header row, which is the first row, is skipped in order to concentrate on the numerical values.

The program then progresses to the second column of the CSV file under the expectation that it will be numerical data to be calculated. The function is versatile enough to handle any kind of data, provided the numerical data is placed in the second column, and can be easily modified to a different kind of input if necessary. Input data are scanned line by line, and every value is converted into a floating-point number to allow precise mathematical operation. The operation identifies non-numeric values as errors and does not add them to the program's output.

The program then computes two important values: the Total Sum and the Average. The Total Sum is computed using Python's optimized built-in sum() function, which is specifically designed for the purpose. It simply adds up all the numeric values parsed from the CSV file. The Average is computed by dividing the Total Sum with the number of data entries. To avoid any mistakes when operating on an empty dataset, the code initially checks if data had been read prior to division. If the dataset is empty, the program sets the average to a default value of zero. This is extremely important in avoiding mistakes like division by zero, where the program will crash. To avoid instances like these from working correctly, Python's exception handling mechanism is utilized. The try-except blocks enable the program to elegantly catch any calculation or data reading error, and this is the way one ought to do errors in Python when dealing with possibly erroneous user inputs and file I/O operations (Downey, 2015).

Once the computations are done, the results are saved to an output file whose name is also provided as a command-line argument. This provides the user with the flexibility of indicating where they wish to store the results. The program employs Python's file handling feature, i.e., the with open() syntax, which takes care of ensuring that the file gets properly opened and closed without any explicit closing. This is best practice for file handling since it does not leave files open unintentionally and thus lead to memory leaks or data corruption. Output is human-readable with a clear display of Total Sum and Average values.

For using memory in the most efficient way, particularly for large datasets, the program reads the input file line by line. In this way, it does not read all data at once, which in the case of large files would be inefficient. The program reads the file incrementally and processes the line as and when it is read. This organization enables better memory handling and enables the program to work well with even more sets of data. This incremental processing and memory optimization mechanism is quite debated in the world of Python programming, for instance, in the tutorial "Efficiency in Python Programming" by Lutz (2013). The technique keeps the program efficient even when it deals with extremely big sets of data, which otherwise would lead to performance degradation.

This clear structure ensures the program is effective and easy to understand. Each step in the program is thought out to meet the needs of the project with software engineering methods that emphasize simplicity, effectiveness, and error handling. The modularity of the program ensures that it is easy to change and add to in the future, should additional features or improvements be needed. This approach helps maintain high-quality code as it makes the program run efficiently based on its purpose.

**Key Technical Breakdown**

* **Error Handling**: Using try-except blocks, the program can handle various errors like missing files or invalid data formats.
* **Command-Line Arguments**: sys.argv is used to accept file paths for input and output files, providing flexibility.
* **Code Optimization**: The program processes the data incrementally, avoiding the use of unnecessary memory-intensive operations.
* **Commenting**: Each function in the program is well-commented, ensuring clarity in the code and assisting future development or debugging.

#### Reflective Evaluation

The activity of designing this Data Processing program served to enhance the understanding of how to apply Python in real life to process real data. Dealing with CSV data and ensuring that the program could deal with lots of data at a time was especially intriguing. I have learned to use Python's internal modules such as csv to parse data and handling different forms of input data through the application of error handling techniques to prevent the program from being terminated in case of any unexpected file formats.

The most critical challenge was ensuring that the calculation (average and sum) was correctly executed, particularly when it came to edge situations like empty data or wrong data. The presence of these checks helped improve my understanding of Python exception handling and file operations. As far as enhancements in areas, I can enhance the program by adding more sophisticated calculations or improving performance when handling even larger data sets, maybe through the use of libraries like pandas for efficient data handling. Additionally, I can add logging to keep track of any issues occurring while processing, which would help in debugging and tracking processing history.

This project also instilled in me the importance of testing with different sets of data. In the future, I would emphasize the development of an extensive testing framework in order to automate tests for the correctness and reliability of my program such that it is able to handle different real-life situations.

#### 

#### References

* Downey, A. (2015). *Think Python: How to Think Like a Computer Scientist* (2nd ed.). O'Reilly Media.
* Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media.
* Python Software Foundation. (2021). *Python Documentation*. Available at:<https://docs.python.org/3/>

#### Appendix 1: Source code

import sys

import csv

def read\_input\_file(file\_name):

    """Reads data from the input file (CSV format)."""

    try:

        data = []

        with open(file\_name, 'r') as file:

            reader = csv.reader(file)

            # Skip header

            next(reader)

            # Read data values (assuming numerical data in the second column)

            for row in reader:

                try:

                    data.append(float(row[1]))  # Assuming the second column holds the numerical values

                except ValueError:

                    print(f"Error: Invalid data found in row: {row}")

        return data

    except FileNotFoundError:

        print(f"Error: The file {file\_name} was not found.")

        sys.exit(1)

def perform\_calculations(data):

    """Performs two calculations on the input data."""

    # Calculation 1: Sum of all the numbers

    total\_sum = sum(data)

    # Calculation 2: Average of the numbers

    average = total\_sum / len(data) if data else 0

    return total\_sum, average

def write\_output\_file(output\_file, total\_sum, average):

    """Writes the results to an output file."""

    try:

        with open(output\_file, 'w') as file:

            file.write(f"Total Sum: {total\_sum}\n")

            file.write(f"Average: {average}\n")

    except IOError:

        print(f"Error: Unable to write to the file {output\_file}.")

        sys.exit(1)

def main():

    if len(sys.argv) < 3:

        print("Usage: python data\_processing.py <input\_file> <output\_file>")

        sys.exit(1)

    input\_file = sys.argv[1]

    output\_file = sys.argv[2]

    # Read input data

    data = read\_input\_file(input\_file)

    # Perform calculations

    total\_sum, average = perform\_calculations(data)

    # Write output data

    write\_output\_file(output\_file, total\_sum, average)

    print(f"Results have been written to {output\_file}")

if \_\_name\_\_ == '\_\_main\_\_':

    main()

#### Appendix 2: Script file

**Terminal command:** .\run\_data\_processing.bat input\_data.csv output\_results.txt

@echo off

REM run\_data\_processing.bat – Script to launch the Data Processing program

REM Usage: run\_data\_processing.bat input\_data.csv output\_results.txt

IF "%~1"=="" (

    ECHO Usage: %~nx0 input\_data.csv output\_results.txt

    EXIT /B 1

)

REM Call Python script with supplied arguments

py data\_processing.py "%~1" "%~2"