Lab 3: Grade Calculator II

Authors: Evan Conley, Brandon Burton

April 23, 2018

1. Problem Statement

Refactor the grade calculator program so that it utilizes file I/O to take graded input and output a summary of graded sections in an html file.

Requirements:

* The program shall take graded input from file.
* The program shall output a grade summary in a .html file called grades.html
* The program shall gracefully handle input errors from an ill-formatted
* The program shall remove additional whitespace between items of each input line before parsing the data.

Assumptions:

* Lines from the input file will be of the same form:
  + AssignmentType Weight% score1/total1, score2/total2, … , scoren/totaln
  + ex.) Homework 10% 20/40, 40/40, 20/25, 100/100
* A missing value of points possible on an assignment implies that the points gained are extra credit (i.e. The total points weighted towards the grade is treated as 0).
* A user can enter the name of which input file to utilize.
* The data from this file should be parsed and separated by grade classification/type (determined by a String at the beginning in each line).
* It is assumed that the user only requires one potential output stream; so, the program stores to a file with a fixed name (grades.html).

Constraints:

* Program cannot create two distinct assignment categories of the same name.
* Program cannot handle non-plaintext formatted inputs (i.e. cannot handle Word Documents).

2. Planning

Upon discussing the potential implementation of file I/O for the program, we decided to implement two functions; one to parse through the input file, cleaning and collecting data, and one to output the graded summary to the grades.html file. As to storage of the grade data from our input, we chose to create a dictionary of dictionaries. The name of the assignment type acted as a primary key. The weight subdictionary stored a single float value, and the scores/maxes dictionaries contained lists of the collected score values. In storing the scores/maxes subdictionaries with values of lists, we did not have to modify the functionality of code from previous lab, making output a lot easier to handle.

NOTE: The following information contains only the set of functions implemented for this lab. For information regarding previously written function definitions, please refer back to CS 356 Lab One Documentation.

Signature: **file\_parser(fh, dictionary)**

Input: The file handle (fh) of the input and the dictionary (dictionary) to be populated with graded

input information.

Output: None

Description: This function will parse through an input file of grade information and populate a

dictionary with the parsed data.

Algorithm: For each line in the file, strips any additional whitespace from left/right of line

Removes whitespace greater than one space between items

Splits data into list elements based on character delimiters

Stores data from each list as a key:value pair in the passed in dictionary

Signature: **file\_output(dictionary)**

Input: Takes the grade information (dictionary)

Output: None

Description: This function will format and store information from graded dictionary to an .html

file.

Algorithm: Create a file handle for ‘grades.html’

Open ‘grades.html’ as a writable file.

For line in dictionary, format the assignment type, weight, average grade, and

weighted grade in a string of the form of html tags.

After this, add final score and letter grade to output string.

Write the output string to grades.html.

Close the file handle.

Signature: **main()**

Input: None

Output: None

Description: This function will be used to test the functionality of the newly implemented

functions on an input file.

Algorithm: Request the name of an input file to be parsed from user.

Try to open the file.

If successful, initialize empty grade dictionary.

Parse through the input file and collect grade information.

For grade category in the grades dictionary, find the average and weighted grade

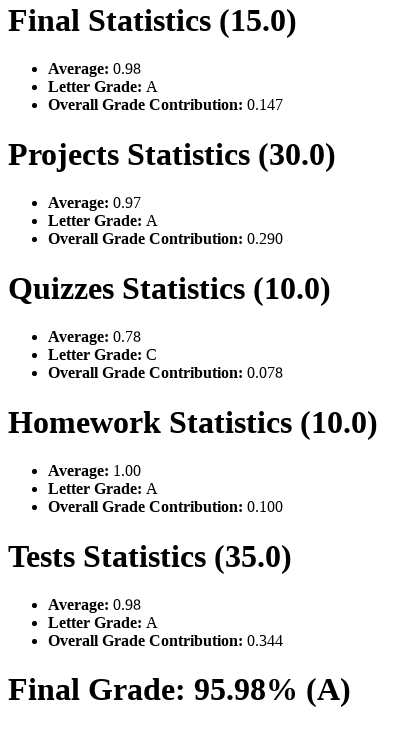
for the category.

Write the information to an html file as an output.

3. Implementation and Testing

After deciding upon the use of a nested dictionary architecture, we decided to utilize the name of each assignment category as the primary key dictionary. From here, we could use sub-keys to gain access to information related to each category. In order to handle this, we created the file\_parser function such that it created the subkey fields and store the data parsed by the function. The main function asks for the name of a file to utilize as seen in Figure 1. Following the use of the file\_parser function, the dictionary populated by the function is passed to the file\_output function. This outputs the data associated with each primary key in the dictionary to the grades.html file. The formatted html output is then parsed and displayed as seen in Figure 2.



**Figure 1. Program Execution**

**Figure 2. Program Output**

4. Reflection and Refactoring

Our initial implementation of the parser function worked only with very well-formatted input based on our assumptions of what the input would look like. The output function, though, worked relatively well as long as the file\_parser function was able to complete. For this reason, very little change was made to the output function after testing with the other group.

That being said, after allowing the other team to change the input in order to see what kind of code-breaking errors could be created, the file\_parser function had to be changed markedly. We created a list of the faults found in our program by the other team; at this point, we utilized these as test cases in order to see if we were able to handle ill-formatted or unexpected inputs. Whitespace removal was handled with regex command and the functions rstip() and lstrip(). Then, any additional or unexpected symbols were stripped using the format command on each element of the grades section of the line. Further, the lines were split on expected values.

Both teams utilized try/except attempts in the parser function, but each handled specific errors differently. Although both solutions seemed to handle a fair amount of unexpected inputs, we believe our solution had a bit of an edge over the other teams. The other team did not utilize regex or strip() functions in combination, adding more specific if/else checks and, overall, somewhat decreasing the readability of the solution. Also, we noted their use of lists within a dictionary as the means of gathering the data from file. It read differently, but functioned quite similarly to our solution in regards to storage.

Looking ahead, though we handled quite a bit of errors, the code could be refactored to be more robust. For example, being able to handle missing values through a try/except that could continue despite the ill-formatted input, rather than just failing gracefully. Further, we believe that we could have created a better namespace in regards to our implemented functions and their parameters. For example, file\_output could have been called html\_file\_creator and, in both our parser and output function, we should have named the dictionary something less generic (i.e. grades\_dict or something of that fashion). Overall, though, we were happy with our solution and our ability handle identified errors from our fellow team members.