Assignment 1 Solution

Raymond Tu tur1

January 25, 2019

Part 2 of Assignment 1, discussing testing both my own modules as well as my partner's modules, any assumptions made during test and possible improvements to the design.

1 Testing of the Original Program

1.1 General Rationale for Test Cases

In general, my approach to the test cases were to have a general base case, that is a case with normal inputs as could be expected when the modules are implemented in real life. Afterwards, the latter test cases are boundary test cases made to test potential unaccounted for errors in the modules, while these test cases are unlikely to happen in actual use, proving that the modules work for these boundary cases proves the robustness of the code. If the module passes these 2 kinds of test cases, then it helps reinforce the robustness and reliability of the code.

1.2 Test Cases for sort(S)

For the first test case for the sort(S) function, it was a general base case with arbitrary macIDs and names, and GPAs within a range of 1 to 12. The second test case was a boundary case, consisting of GPAs that had duplicates, and only had the values of 1 and 0. This was chosen to see how the function responded to sorting the boundary case of a GPA of 1 or 0, and to see how it would sort duplicates, with the assumption that the entry that came first in the input list would be sorted first in the case of a duplicate. The third test case is a boundary test case where the input is an empty list, with the assumption that it would return an empty list.

1.3 Test Cases for average(L, g)

The majority of the test cases for the average(L, g) function were tested for both male and female as input parameter for gender, this will be considered 1 test case. Additionally, an assumption made for the average function is that the returned average GPA would be rounded to a single decimal place, much like the one's on our transcript. The first test case for the average(L, g) function was a base case, with arbitrary student details and GPAs within a range of 1 to 12. The second test case is with multiple GPAs of 1 or 0 to test the program's accuracy with results that are less than 1. The third test case consisted of an empty list for a boundary case to see if the program correctly returns 0 instead of attempting to a division over 0. The fourth test case only has GPAs with 0 to see if the program correctly returns 0 as a boundary case.

1.4 Test cases for allocate(S, F, C)

For the allocate function several assumptions were made. The first assumption is that even if a student has free choice, and if their GPA is below 4, they will not be allocated. The second assumption made was that free choice students will receive their first choice regardless of circumstances, even if there is not enough department capacity. The department capacity is checked after allocating the free choice students, and if a department is overloaded, then it is considered to be full (ie. has negative seats left). The third assumption made is that all students with free choice are taken from the list of student dictionaries passed into the program. The last assumption made is that any students who were not allocated would ideally be given a choice to enter any department that still has spots, however this is out of the scope of the function.

The first test case for the allocate function was a general base case, with reasonable department capacities and varied free choice students. The second test case was a boundary test case where an empty list of student dictionaries is passed in, expecting a returned empty dictionary. The third test case is a boundary test case where all the departments are full and there are no free choice students, to see if the program correctly does not allocate any students. The final test case is to see if it follows the assumption made that free choice students are allocated even though there are not enough department spaces for them.

1.5 Results of Testing Original Code

The final rendition of the CalcModule.py passed all of my test cases without issue. However, prior to it there were several problems encountered requiring the module be changed to account for these boundary cases. While the majority of the boundary cases passed without issue for the sort and allocate function, there were errors encountered when testing the average function with test cases 2 and 3. In the second test case, the function had errors in returning a correct average due to the inaccuracy of floating point numbers. In the third test case, the function originally did not take into account the potential case for an empty list leading to a division over 0.

Additionally, in my renewed testing I realized my test case expected output for the first test case for the allocate function was incorrect, resulting in not catching the critical error of the free choice students being allocated twice. This error was changed in the testCalc.py before testing my partner files.

2 Results of Testing Partner's Code

Without any modifications to my partner's code, the test module failed to run due to their module being reliant on importing modules outside of the CalcModule.py, namely a1_constants and a1_utility. In an attempt to fix my partner's code to run my test cases, I commented out the import statements and made assumptions of what they intended to import based on the variable names. The 2 values changed throughout my partner's code were the 2 constants imported, MIN_PASSING_GPA and MAX_GPA. I assumed they were referring to the minimum GPA to pass and the maximum possible GPA, so I replaced these constants with the values 4 and 12 respectively. Additionally, I commented out the called function remove_duplicate(students) line that was calling a function from the a1_utility module. After these changes, the testCalc.py ran successfully without syntax errors.

For the sort(S) function, all 3 of my test cases passed. On the other hand, all 4 of the test cases for the average function failed. For the allocate function, test cases 1 and 4 failed while 2 and 3 passed. Potential reasons for these failed test cases and problems in code will be discussed in the next section.

3 Discussion of Test Results

3.1 Problems with Original Code

During my own testing process, there were no problems in the sort(S) function, however there were originally issues with the average(L, g) function. Test cases 2 and 3 failed, which were a result of the program not taking into account the boundary cases and incorrectly rounding the result of a floating point calculation. The first problem was fixed in the code by adding a conditional if statement to check if the given input parameter for list L was empty, if true then the program would return 0 rather than attempting a division over 0. The second problem was fixed by making the assumption that the calculated average GPAs would be rounded to a single decimal place similar to our transcripts. This fixed issue because within a single decimal place float calculations remain accurate.

Additionally, while testing my own code again after having my partner's code, I've realized there is a critical error in how I deleted the allocated free choice students from my list, with del students being incorrect for my intended purpose. This resulted in free choice students being allocated twice, resulting in the first test case failing because the free choice students were not being deleted from the list. I fixed this issue to changing it to elgible_std.remove(students), this correctly removes the allocated free choice students from the list so they are only allocated once.

3.2 Problems with Partner's Code

3.3 Problems with the average(L, g) Function

For my partner's code, all 4 of the test cases for the average(L, g) failed. After printing out the output of each case to attempt to determine the issue, one problem that I found was that we made different assumptions for the rounding of the GPA. While I rounded the GPA to a single decimal place for increased accuracy due to the problems with calculations using floating point numbers, my partner didn't, resulting in all the test cases failing because the decimal places weren't accurate. A second potential issue with the average function is the way the average was calculated. Rather than storing the sum of the GPAs in a float or double variable, they instead decided to append these GPAs into a list and calculate the sum of the list divided by the length of the list to determine the average. This could potentially introduce errors when appending GPAs based on gender into the list, causing the average to be inaccurate.

3.4 Problems with the allocate(S, F, C) Function

For the allocate(S, F, C) function test cases 1 and 4 failed, which was the base case and the boundary case with the assumption that all free choice students are enrolled in their first choice regardless of department capacity. Test cases 2 and 3 passed, which were both boundary cases, test case 2 passed in an empty list of student dictionaries while test case 3 allocated no students due to all departments being full. This mean's that the function does correctly take into account the department capacities but is not allocating the students correctly due to the base case failing. The biggest potential factor in the failure to correctly allocate students may be due to my own modification to comment out the remove_duplicate(students) line that was imported from their a1_utility module. This modification as mentioned previously was done in order to have the program successfully compile. With the assumption that this helper function helped the function sort lists with duplicate macIDs, then the lack of this function would cause it to fail the test cases as I did have duplicate students at times in my own allocate before fixing the aforementioned error. For test case 4, the function failed likely due to the differing assumptions made, while I made the assumption that free choice students would be allocated regardless of department capacity. This is likely not due to an error in my partner's code, but instead the problems of natural language specification.

4 Critique of Design Specification

In my opinion, the natural language design specification offered lots of free form creativity and interpretation of how to solve the problem, however this resulted in many issues and confusion regardless some of the details of the modules. While the openness of the specification allowed for creativity, I would improve the specification by making some of the requirements clearer and less open to interpretation, or in other words less assumptions needing to be made. For example, specifying some details such as how many decimal places to round the GPA average to, whether free choice students can exceed department capacity, and what to do with unallocated students would greatly clarify some of the doubts had during the assignment. Additionally, having sample outputs listed for each of the modules would greatly increase one's own confidence in the correctness of the code when testing.

5 Answers to Questions

(a) The current average function is too specific in only calculating the average GPA for one gender in the student dictionary. To make the function more general, the function could take in a third valid g parameter as 'none', when this is specified than instead of calculating the average of a specific gender, it would calculate the average for all students. Additionally, a third parameter could be added to take in a list of free choice students, and when this parameter is passed in the function would compute the average of free choice students. Along the same line of thinking, additional parameters could be added to the sort(S) function to give it more options than sorting in descending order. For example a second parameter a_or_d that would specify whether the function will sort in ascending or descending order. An additional change similar to the average function would be adding a parameter to take in a list of free choice students, to only sort the free choice students in the student dictionary.

(b) In this context, aliasing in python refers to when a new variable is given the value of another variable, effectively becoming an 'alias' for that variable, where when one variable is changed the other is changed as well. Aliasing can definitely be an issue for dictionaries, because you may alias a key or it's value, leading bugs and confusion when trying to trace the program to find the source of an error. You might guard against these problems by making sensitive key values in the program immutable, or using a similar data type that is immutable, such as a tuple.

(c) Some potential test cases for ReadAllocationData.py, would be a general base case for each of the text files and then boundary cases. These boundary cases could consist of empty text files, or text files in an incorrect format to the one assumed, for example for the readFreeChoice(s) one boundary case could be the free choice students being listed on a single line instead of the imagined every other line format. CalcModule.py was likely selected over ReadAllocationData.py as the module to be tested because CalcModule.py had more intensive functions and more sources of error. CalcModule.py had many sources of errors, such as floating point inaccuracy and correctly allocating the students with the multitude of different factors and scenarios, and with this more assumptions had to be made. Because of the increased complexity, it lead to the testCalc.py module to be more intensive overall.

(d) The problem with using strings in this way is that there can easily be typos in the keys, or different interpretations of capitalization that would lead to the module being unable to be used by someone else. For example, male can be written as male, Male, MALE, etc. A potentially better way is to use a condensed form, such as a capital G to represent the key gender, or to create a an object such as a class to represent the gender key.

(e) Other ways to implement records and structs in python besides a dictionary are tuples, named tuples, custom classes or the structures class. However, even with all these options, I would not recommend changing the data structures used in the code modules for this assignment. This is due to the immutability of the tuples and named tuples, while preventing typos it would make the implementation of the allocate function increasingly difficult if every student could only be accessed but not changed. On the other hand, a custom class is even more flexible than dictionaries and may lead to problems with aliasing and other bugs when working with this data structure. If I were to change the data structure of the code modules, I would use a named tuple to have similar readability to dictionaries with the added benefit of immutability.

(f) If the list of strings was changed to a different data structure, such as a tuple then the CalcModule.py would have to be changed accordingly because accessing an element from a tuple is different from accessing an element in a list. If instead a custom class was created for the students and CalcModule.py was correctly modified to call the class, then CalcModule.py does not need to be modified again if the class structure changes. This is because the module would only be calling methods in the class, and the syntax to call these methods does not change based on the structure of the class, the methods would still return the next choice (ie. a string value chemical, mechanical, software etc.) or the boolean value true when there are no more choices. Both the syntax to call the methods of the class as well as the returned values stay the same, so there would be no need to modify CalcModule.py if the class data structure changes. This demonstrates the benefits of modularity in a program.

F Code for ReadAllocationData.py

```
## @file ReadAllocationData.py
#  @brief Module that contains the functions readStdnts(s), readFreeChoice(s), and
     readDeptCapacity(s). All 3 take in a given .txt file to generate the desired output into a list
     or a dictionary.
#  @author Raymond Tu
#  @date 2019-01-18

## @brief Takes in a text file to create a list of student dictionaries.
#  @details Creates a list of student dictionaries where each entry in the list is one student, with
     all their details such as macid, first and last name, gpa as well as their first, second and
     third choice of program.
#  @param s A string for the name of the text file that contains student information on every line, in
     order of macid, name, gender, gpa and program choice respectively.
#  @return A list of student dictionaries generated from the passed in text file.
def readStdnts(s):

        file = open(s, 'r')
        lines = file.readlines()

        std_list = []

        for line in lines:

                print(line)
                print(line.split())
                split_list = line.split()

                std_dict = {'macid': 's', 'fname': 's', 'lname': 's', 'gender': 's', 'gpa': 's',
                     'choices': 's'}
                std_dict['macid'] = split_list[0]
                std_dict['fname'] = split_list[1]
                std_dict['lname'] = split_list[2]
                std_dict['gender'] = split_list[3]
                std_dict['gpa'] = split_list[4]
                choice_list = []
                choice_list.append(split_list[5])
                choice_list.append(split_list[6])
                choice_list.append(split_list[7])
                std_dict['choices'] = choice_list

                std_list.append(std_dict)

        for std in std_list:
                print(std)

        print(std_list)

        file.close()

        return std_list

#test.txt used to test readStdnts

#readStdnts("test.txt")

## @brief Takes in a text file to create a list of macids for students with free choice.
#  @details Takes in a text file where every line is a macid of a student with free choice and creates
     a list out of them where each entry is a macid.
#  @param s A string for the name of the text file that the macids for students with free choice.
#  @return A list of mac ids of students with free choice.
def readFreeChoice(s):


        file = open(s, 'r')
        content = open(s, 'r').read()


        print(content)

        free_choice = content.split()

        print(free_choice)

        file.close()

        return free_choice
```

```
#test1.txt used to test readFreeChoice
#readFreeChoice("test1.txt")

## @brief Takes in a text file to create a dictionary of department capacities.
#  @details Creates a list of student dictionaries from a text file where every line is the department
     followed by the number of spots in their respective department. The function takes this file and
     creates a dictionary out of this, where each key corresponds to a department.
#  @param s A string for the name of a text file that contains a department followed by it's capacity
     on every line.
#  @return A dictionary of department capacities generated from text file of string s.
def readDeptCapacity(s):

        file = open(r'C:\Users\Raymond\Documents\2AA4\tur1\A1\src\test2.txt', 'r')
        lines = file.readlines()

        dept_dict = {'civil': 'i', 'chemical': 'i', 'electrical': 'i', 'mechanical': 'i', 'software':
             'i', 'materials': 'i', 'engphys': 'i'}

        counter = 0

        for line in lines:
                dept = line.split()

                if counter == 0:
                        dept_dict['civil'] = dept[1]
                elif counter == 1:
                        dept_dict['chemical'] = dept[1]
                elif counter == 2:
                        dept_dict['electrical'] = dept[1]
                elif counter == 3:
                        dept_dict['mechanical'] = dept[1]
                elif counter == 4:
                        dept_dict['software'] = dept[1]
                elif counter == 5:
                        dept_dict['materials'] = dept[1]
                elif counter == 6:
                        dept_dict['engphys'] = dept[1]

                counter = counter + 1

        print(dept_dict)

        return dept_dict

#test2.txt used to test readDeptCapacity
#readDeptCapacity("test2.txt")
```

G Code for CalcModule.py

```
## @file CalcModule.py
#  @brief Module that contains the functions sort, average and allocate that take the generated
     dictionaries and lists from Step 1.
#  @author Raymond Tu
#  @date 2019-01-18

import operator

## @brief Function that sorts the given list of dictionaries in descending order by GPA.
#  @details Uses the operator library to use the inbuilt sort function to sort the list of
     dictionaries S, by GPA in descending order.
#  @param S The given list of student dictionaries generated from readStdnts(s) in
     ReadAllocationData.py
#  @return Sorted list of student dictionaries in descending order by GPA.
def sort(S):

#code taken from
     https://stackoverflow.com/questions/72899/how-do-i-sort-a-list-of-dictionaries-by-a-value-of-the-dictionary
        from operator import itemgetter
        sorted_list = sorted(S, key=itemgetter('gpa'), reverse= True)

        return sorted_list

## @brief Function that calculates the average GPA for male or female students.
#  @details Loops through every male or female in the list of student dictionaries, adds up all the
     GPAs then divides by number of students to determine average.
#  @param L The given list of student dictionaries generated from readStdnts(s) in
     ReadAllocationData.py
#  @param g The desired gender of students to find the average GPA from.
#  @return The average of male or female students in the list of student dictionaries.
def average(L, g):

        male_list = []
        female_list = []

        if len(L) == 0:
                return 0

        for students in L:
                if students['gender'] == 'male':
                        male_list.append(students)
                elif students['gender'] == 'female':
                        female_list.append(students)

        if len(male_list) == 0:
                return 0

        if len(female_list) == 0:
                return 0

        num_males = 0
        num_females = 0
        male_total = 0
        female_total = 0
        male_average = 0
        female_average = 0

        if g == 'male':
                for males in male_list:
                        male_total = male_total + males['gpa']
                        num_males = num_males + 1
                male_average = male_total/num_males
                #assumption that the average GPA will be rounded to 1 decimal place
                rounded_male_average = round(male_average, 1)
                return rounded_male_average

        elif g == 'female':
                for females in female_list:
                        female_total = female_total + females['gpa']
                        num_females = num_females + 1
                female_average = female_total/num_females
                #assumption that the average GPA will be rounded to 1 decimal place
                rounded_female_average = round(female_average, 1)
                return rounded_female_average

## @brief Function that allocates students to their choice of program.
```

```
#  @details Function that allocates students to their choice of program, where students with free
     choice are given priority, followed by GPA.
#  @param S The given list of student dictionaries generated from readStdnts(s) in
     ReadAllocationData.py
#  @param F List of students with free choice by macID generated from function readFreeChoice.
#  @param C A dictionary of department capacities generated by readDeptCapacity.
#  @return The allocated list of students, listing every department and the students enrolled in them.
def allocate(S, F, C):
        elgible_std = []

        #assumption that students with a GPA of exactly 4.0 will still be allocated
        #assumption that even if student has free choice, if GPA is lower than 4 not elgible to be
             allocated
        for students in S:
                if students['gpa'] >= 4.0:
                        elgible_std.append(students)

        allocated_list = {'civil':[], 'chemical':[], 'electrical':[], 'mechanical':[], 'software':[],
             'materials':[], 'engphys':[]}

        #assumption that free choice students will receive their first choice no matter the
             circumstances, for example if the number of free choice students exceeds department
             capacity
        for free_students in F:
                for students in elgible_std:
                        if students['macid'] == free_students:
                                allocated_list[students['choices'][0]].append(students)
                                #once allocated free choice students will not be allocated again,
                                     removed from general list
                                elgible_std.remove(students)

        #after free choice students are allocated, allocate students based on highest to lowest GPA
        from operator import itemgetter
        sorted_list = sorted(elgible_std, key=itemgetter('gpa'), reverse= True)

        #Assumption that number of spots left are only checked after free choice is allocated, so
             potential for "negative capacity", in which case the department is said to be full
        civil_spots = C['civil'] - len(allocated_list['civil'])
        chem_spots = C['chemical'] - len(allocated_list['chemical'])
        elec_spots = C['electrical'] - len(allocated_list['electrical'])
        mech_spots = C['mechanical'] - len(allocated_list['mechanical'])
        soft_spots = C['software'] - len(allocated_list['software'])
        mat_spots = C['materials'] - len(allocated_list['materials'])
        engphys_spots = C['engphys'] - len(allocated_list['engphys'])

        for students in sorted_list:
                first_choice = students['choices'][0]
                second_choice = students['choices'][1]
                third_choice = students['choices'][2]

                if (first_choice == 'civil') and (civil_spots > 0):
                        allocated_list['civil'].append(students)
                        civil_spots = civil_spots - 1
                elif (first_choice == 'chemical') and (chem_spots > 0):
                        allocated_list['chemical'].append(students)
                        chem_spots = chem_spots - 1
                elif (first_choice == 'electrical') and (elec_spots > 0):
                        allocated_list['electrical'].append(students)
                        elec_spots = elec_spots - 1
                elif (first_choice == 'mechanical') and (mech_spots > 0):
                        allocated_list['mechanical'].append(students)
                        mech_spots = mech_spots - 1
                elif (first_choice == 'software') and (soft_spots > 0):
                        allocated_list['software'].append(students)
                        soft_spots = soft_spots - 1
                elif (first_choice == 'materials') and (mat_spots > 0):
                        allocated_list['materials'].append(students)
                        mat_spots = mat_spots - 1
                elif (first_choice == 'engphys') and (engphys_spots > 0):
                        allocated_list['engphys'].append(students)
                        engphys_spots = engphys_spots - 1
                elif (second_choice == 'civil') and (civil_spots > 0):
                        allocated_list['civil'].append(students)
                        civil_spots = civil_spots - 1
                elif (second_choice == 'chemical') and (chem_spots > 0):
                        allocated_list['chemical'].append(students)
                        chem_spots = chem_spots - 1
                elif (second_choice == 'electrical') and (elec_spots > 0):
                        allocated_list['electrical'].append(students)
                        elec_spots = elec_spots - 1
                elif (second_choice == 'mechanical') and (mech_spots > 0):
                        allocated_list['mechanical'].append(students)
                        mech_spots = mech_spots - 1
                elif (second_choice == 'software') and (soft_spots > 0):
                        allocated_list['software'].append(students)
                        soft_spots = soft_spots - 1
                elif (second_choice == 'materials') and (mat_spots > 0):
                        allocated_list['materials'].append(students)
                        mat_spots = mat_spots - 1
                elif (second_choice == 'engphys') and (engphys_spots > 0):
                        allocated_list['engphys'].append(students)
                        engphys_spots = engphys_spots - 1
                elif (third_choice == 'civil') and (civil_spots > 0):
                        allocated_list['civil'].append(students)
                        civil_spots = civil_spots - 1
                elif (third_choice == 'chemical') and (chem_spots > 0):
                        allocated_list['chemical'].append(students)
                        chem_spots = chem_spots - 1
                elif (third_choice == 'electrical') and (elec_spots > 0):
                        allocated_list['electrical'].append(students)
                        elec_spots = elec_spots - 1
                elif (third_choice == 'mechanical') and (mech_spots > 0):
                        allocated_list['mechanical'].append(students)
                        mech_spots = mech_spots - 1
                elif (third_choice == 'software') and (soft_spots > 0):
                        allocated_list['software'].append(students)
                        soft_spots = soft_spots - 1
                elif (third_choice == 'materials') and (mat_spots > 0):
                        allocated_list['materials'].append(students)
                        mat_spots = mat_spots - 1
                elif (third_choice == 'engphys') and (engphys_spots > 0):
                        allocated_list['engphys'].append(students)
                        engphys_spots = engphys_spots - 1

        #assumption that after all students have been allocated, any students who were not able to be
             allocated in their first, second or third choice will be given an option to enter a
             department with open capacity, this is out of the scope of this function.

        return allocated_list
```

H Code for testCalc.py

```
## @file testCalc.py
#  @title testCalc
#  @author Raymond Tu
#  @date 2019-01-18

## @brief This module is for testing the functions in CalcModule.py.

import CalcModule

#Test case 1 for sort(S)

##First test case for sort(S) function
#First test case is a base case with regular GPA values and arbitrary macid
S_test1 = [{'macid': 'frank1', 'fname': 'frank', 'lname': 'jill', 'gender':
'male', 'gpa': 8.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'bob1', 'fname':
     'bob', 'lname': 'jill', 'gender':
'male', 'gpa': 11.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'tur1', 'fname':
     'raymond', 'lname': 'tu', 'gender':
'male', 'gpa': 10.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'jill1', 'fname':
     'jill', 'lname': 'jill', 'gender':
'male', 'gpa': 9.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'sally', 'fname':
     'sally', 'lname': 'jill', 'gender':
'male', 'gpa': 7.0, 'choices': ['chemical', 'mechanical', 'software']}]

##First expected output for sort(S) function
#Students sorted by GPA in descended order
expected_output1 = [{'macid': 'bob1', 'fname': 'bob', 'lname': 'jill', 'gender':
'male', 'gpa': 11.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'tur1', 'fname':
     'raymond', 'lname': 'tu', 'gender':
'male', 'gpa': 10.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'jill1', 'fname':
     'jill', 'lname': 'jill', 'gender':
'male', 'gpa': 9.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'frank1', 'fname':
     'frank', 'lname': 'jill', 'gender':
'male', 'gpa': 8.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'sally', 'fname':
     'sally', 'lname': 'jill', 'gender':
'male', 'gpa': 7.0, 'choices': ['chemical', 'mechanical', 'software']}]
##Output of first test case
output = CalcModule.sort(S_test1)

#print(output)

if output == expected_output1:
        print("Test case 1 for sort(S) passed")
else:
        print("Test case 1 failed for sort(S)")

#Test case 2 for sort(S)
#Test case created to test how it sorts with multiple entries students with the same GPA, as well how
     it handles the boundary
#case with 0s and 1s
S_test2 = [{'macid': 'frank1', 'fname': 'frank', 'lname': 'jill', 'gender':
'male', 'gpa': 0.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'bob1', 'fname':
     'bob', 'lname': 'jill', 'gender':
'male', 'gpa': 1.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'tur1', 'fname':
     'raymond', 'lname': 'tu', 'gender':
'male', 'gpa': 1.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'jill1', 'fname':
     'jill', 'lname': 'jill', 'gender':
'male', 'gpa': 0.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'sally', 'fname':
     'sally', 'lname': 'jill', 'gender':
'male', 'gpa': 0.0, 'choices': ['chemical', 'mechanical', 'software']}]

expected_output2 = [{'macid': 'bob1', 'fname': 'bob', 'lname': 'jill', 'gender':
'male', 'gpa': 1.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'tur1', 'fname':
     'raymond', 'lname': 'tu', 'gender':
'male', 'gpa': 1.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'frank1', 'fname':
     'frank', 'lname': 'jill', 'gender':
'male', 'gpa': 0.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'jill1', 'fname':
     'jill', 'lname': 'jill', 'gender':
'male', 'gpa': 0.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'sally', 'fname':
     'sally', 'lname': 'jill', 'gender':
'male', 'gpa': 0.0, 'choices': ['chemical', 'mechanical', 'software']}]


output2 = CalcModule.sort(S_test2)

#print(output2)
```

```
if output2 == expected_output2:
        print("Test case 2 for sort(S) passed")
else:
        print("Test case 2 failed for sort(S)")


#Test case 3 for sort(S)
#Test case created with empty list, boundary case
S_test12 = []

expected_output12 = []

output12 = CalcModule.sort(S_test12)

#print(output12)

if output12 == expected_output12:
        print("Test case 3 for sort(S) passed")
else:
        print("Test case 3 failed for sort(S)")

#Test case 1 for average(L, g) for male
#Regular base test case, with males, females, GPA within range of 12-0
A_test1 = [{'macid': 'tur1', 'fname': 'bob', 'lname': 'bobbert', 'gender':
'male', 'gpa': 12.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'tur1', 'fname':
     'jill', 'lname': 'jillian', 'gender':
'female', 'gpa': 10.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'tur1', 'fname':
     'bob', 'lname': 'jill', 'gender':
'male', 'gpa': 9.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'tur1', 'fname':
     'sally', 'lname': 'jill', 'gender':
'female', 'gpa': 2.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'tur1', 'fname':
     'rob', 'lname': 'jill', 'gender':
'male', 'gpa': 7.0, 'choices': ['chemical', 'mechanical', 'software']}]

#Calculated average by hand
expected_output3 = 9.3

output3 = CalcModule.average(A_test1, 'male')


if output3 == expected_output3:
        print("Test case 1 for average(L, g) for male passed")
else:
        print("Test case 1 for average(L, g) for male failed")

#Test case 1 for average(L, g) for female
#Same test case but with g as female instead
A_test2 = [{'macid': 'tur1', 'fname': 'bob', 'lname': 'bobbert', 'gender':
'male', 'gpa': 12.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'tur1', 'fname':
     'jill', 'lname': 'jillian', 'gender':
'female', 'gpa': 10.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'tur1', 'fname':
     'bob', 'lname': 'jill', 'gender':
'male', 'gpa': 9.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'tur1', 'fname':
     'sally', 'lname': 'jill', 'gender':
'female', 'gpa': 2.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'tur1', 'fname':
     'rob', 'lname': 'jill', 'gender':
'male', 'gpa': 7.0, 'choices': ['chemical', 'mechanical', 'software']}]

expected_output4 = 6.0

output4 = CalcModule.average(A_test2, 'female')


if output4 == expected_output4:
        print("Test case 1 for average(L, g) for female passed")
else:
        print("Test case 1 for average(L, g) for female failed")

#Test case 2 for average(L, g) for male
#boundary test case with duplicates and 1s and 0s to see if there are any discrepancy in handling
     lower boundary inputs
A_test3 = [{'macid': 'tur1', 'fname': 'bob', 'lname': 'bobbert', 'gender':
'male', 'gpa': 0.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'tur1', 'fname':
     'jill', 'lname': 'jillian', 'gender':
'female', 'gpa': 0.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'tur1', 'fname':
     'bob', 'lname': 'jill', 'gender':
'male', 'gpa': 1.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'tur1', 'fname':
     'sally', 'lname': 'jill', 'gender':
```

```
'female', 'gpa': 1.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'tur1', 'fname':
     'rob', 'lname': 'jill', 'gender':
'male', 'gpa': 0.0, 'choices': ['chemical', 'mechanical', 'software']}]

#Calculated average
expected_output5 = 0.3

output5 = CalcModule.average(A_test3, 'male')


if output5 == expected_output5:
        print("Test case 2 for average(L, g) for male passed")
else:
        print("Test case 2 for average(L, g) for male failed")

#Test case 2 for average(L, g) for female
#Same test base but with g as female instead
A_test4 = [{'macid': 'tur1', 'fname': 'bob', 'lname': 'bobbert', 'gender':
'male', 'gpa': 0.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'tur1', 'fname':
     'jill', 'lname': 'jillian', 'gender':
'female', 'gpa': 0.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'tur1', 'fname':
     'bob', 'lname': 'jill', 'gender':
'male', 'gpa': 1.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'tur1', 'fname':
     'sally', 'lname': 'jill', 'gender':
'female', 'gpa': 1.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'tur1', 'fname':
     'rob', 'lname': 'jill', 'gender':
'male', 'gpa': 0.0, 'choices': ['chemical', 'mechanical', 'software']}]

expected_output6 = 0.5

output6 = CalcModule.average(A_test4, 'female')


if output6 == expected_output6:
        print("Test case 2 for average(L, g) for female passed")
else:
        print("Test case 2 for average(L, g) for female failed")

#Test case 3 for average(L, g) for male
#boundary test case with empty list
A_test5 = []

#Calculated average
expected_output10 = 0

output10 = CalcModule.average(A_test5, 'male')

#print(output10)


if output10 == expected_output10:
        print("Test case 3 for average(L, g) for male passed")
else:
        print("Test case 3 for average(L, g) for male failed")


#Test case 4 for average(L, g) for female
#Boundary test case where all GPA is 0
A_test6 = [{'macid': 'tur1', 'fname': 'bob', 'lname': 'bobbert', 'gender':
'male', 'gpa': 0.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'tur1', 'fname':
     'jill', 'lname': 'jillian', 'gender':
'female', 'gpa': 0.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'tur1', 'fname':
     'bob', 'lname': 'jill', 'gender':
'male', 'gpa': 0.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'tur1', 'fname':
     'sally', 'lname': 'jill', 'gender':
'female', 'gpa': 0.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'tur1', 'fname':
     'rob', 'lname': 'jill', 'gender':
'male', 'gpa': 0.0, 'choices': ['chemical', 'mechanical', 'software']}]

expected_output16 = 0.0

output16 = CalcModule.average(A_test6, 'female')


if output16 == expected_output16:
        print("Test case 4 for average(L, g) for female passed")
else:
        print("Test case 4 for average(L, g) for female failed")
```

```
#Test case 1 for allocate
#Base test case, with free choice students, reasonable department capacity
Allocate_test1 = [{'macid': 'bob1', 'fname': 'bob', 'lname': 'bobbert', 'gender':
'male', 'gpa': 12.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'jill1', 'fname':
     'jill', 'lname': 'jillian', 'gender':
'female', 'gpa': 10.0, 'choices': ['civil', 'engphys', 'software']}, {'macid': 'bob2', 'fname': 'bob',
     'lname': 'jill', 'gender':
'male', 'gpa': 9.0, 'choices': ['engphys', 'mechanical', 'software']}, {'macid': 'sally1', 'fname':
     'sally', 'lname': 'jill', 'gender':
'female', 'gpa': 2.0, 'choices': ['chemical', 'mechanical', 'civil']}, {'macid': 'rob1', 'fname':
     'rob', 'lname': 'jill', 'gender':
'male', 'gpa': 7.0, 'choices': ['civil', 'mechanical', 'software']}]

free_choice = ['bob1', 'bob2', 'rob1']

dept_cap = { 'civil': 1, 'chemical': 2, 'electrical': 0, 'mechanical': 3,  'software': 3, 'materials':
     1,  'engphys': 2}

allocated_list = CalcModule.allocate(Allocate_test1, free_choice, dept_cap)

#print(allocated_list)

expected_output7 = {'civil': [{'macid': 'rob1', 'fname': 'rob', 'lname': 'jill', 'gender':
 'male', 'gpa': 7.0, 'choices': ['civil', 'mechanical', 'software']}], 'chemical': [{'macid': 'bob1',
      'fname': 'bob', 'lname': 'bobbert', 'gender':
 'male', 'gpa': 12.0, 'choices': ['chemical', 'mechanical', 'software']}], 'electrical': [],
 'mechanical': [], 'software': [], 'materials': [], 'engphys': [{'macid': 'bob2', 'fname': 'bob',
      'lname': 'jill', 'gender':
 'male','gpa': 9.0, 'choices': ['engphys', 'mechanical', 'software']}, {'macid': 'jill1', 'fname':
      'jill', 'lname': 'jillian', 'gender':
 'female', 'gpa': 10.0, 'choices': ['civil', 'engphys', 'software']}]}

output7 = CalcModule.allocate(Allocate_test1, free_choice, dept_cap)

if output7 == expected_output7:
        print("Test case 1 for allocate(S, F, C) passed")
else:
        print("Test case 1 for allocate(S, F, C) failed")


#Test case 2 for allocate
#Empty list boundary case

Allocate_test2 = []

free_choice1 = []

dept_cap1 = { 'civil': 1, 'chemical': 2, 'electrical': 0, 'mechanical': 3,  'software': 3,
     'materials': 1,  'engphys': 2}

allocated_list1 = CalcModule.allocate(Allocate_test2, free_choice1, dept_cap1)

#print(allocated_list1)

expected_output8 = {'civil': [], 'chemical': [], 'electrical': [], 'mechanical': [], 'software': [],
     'materials': [], 'engphys': []}

output8 = CalcModule.allocate(Allocate_test2, free_choice1, dept_cap1)

if output8 == expected_output8:
        print("Test case 2 for allocate(S, F, C) passed")
else:
        print("Test case 2 for allocate(S, F, C) failed")

#Test case 3 for allocate
#Boundary test case, no free choice students and no department capacity
Allocate_test3 = [{'macid': 'bob1', 'fname': 'bob', 'lname': 'bobbert', 'gender':
'male', 'gpa': 12.0, 'choices': ['chemical', 'mechanical', 'software']}, {'macid': 'jill1', 'fname':
     'jill', 'lname': 'jillian', 'gender':
'female', 'gpa': 10.0, 'choices': ['civil', 'engphys', 'software']}, {'macid': 'bob2', 'fname': 'bob',
     'lname': 'jill', 'gender':
'male', 'gpa': 9.0, 'choices': ['engphys', 'mechanical', 'software']}, {'macid': 'sally1', 'fname':
     'sally', 'lname': 'jill', 'gender':
'female', 'gpa': 2.0, 'choices': ['chemical', 'mechanical', 'civil']}, {'macid': 'rob1', 'fname':
     'rob', 'lname': 'jill', 'gender':
'male', 'gpa': 7.0, 'choices': ['civil', 'mechanical', 'software']}]

free_choice3 = []
```

```
dept_cap3 = { 'civil': 0, 'chemical': 0, 'electrical': 0, 'mechanical': 0,  'software': 0,
     'materials': 0,  'engphys': 0}

allocated_list3 = CalcModule.allocate(Allocate_test3, free_choice3, dept_cap3)

#print(allocated_list3)


#all depts full and no free choice, shouldn't allocate anyone
expected_output9 = {'civil': [], 'chemical': [], 'electrical': [], 'mechanical': [], 'software': [],
     'materials': [], 'engphys': []}

output9 = CalcModule.allocate(Allocate_test3, free_choice3, dept_cap3)

if output9 == expected_output9:
        print("Test case 3 for allocate(S, F, C) passed")
else:
        print("Test case 3 for allocate(S, F, C) failed")

#Test case 4 for allocate
#Test case with assumption that if students with free choice have first choice that exceeds dept
     capacity, they all are enrolled regardless
Allocate_test4 = [{'macid': 'bob1', 'fname': 'bob', 'lname': 'bobbert', 'gender':
'male', 'gpa': 12.0, 'choices': ['civil', 'mechanical', 'software']}, {'macid': 'jill1', 'fname':
     'jill', 'lname': 'jillian', 'gender':
'female', 'gpa': 10.0, 'choices': ['civil', 'engphys', 'software']}, {'macid': 'bob2', 'fname': 'bob',
     'lname': 'jill', 'gender':
'male', 'gpa': 9.0, 'choices': ['civil', 'mechanical', 'software']}, {'macid': 'sally1', 'fname':
     'sally', 'lname': 'jill', 'gender':
'female', 'gpa': 2.0, 'choices': ['chemical', 'mechanical', 'civil']}, {'macid': 'rob1', 'fname':
     'rob', 'lname': 'jill', 'gender':
'male', 'gpa': 7.0, 'choices': ['civil', 'mechanical', 'software']}]

free_choice4 = ['bob1', 'jill1', 'bob2', 'rob1']

#4 free choice students with civil as first choice but only 2 spots
dept_cap4 = { 'civil': 2, 'chemical': 0, 'electrical': 0, 'mechanical': 0,  'software': 0,
     'materials': 0,  'engphys': 0}

expected_output19 = {'civil': [{'macid': 'bob1', 'fname': 'bob', 'lname': 'bobbert', 'gender':
'male', 'gpa': 12.0, 'choices': ['civil', 'mechanical', 'software']}, {'macid': 'jill1', 'fname':
     'jill', 'lname': 'jillian', 'gender':
'female', 'gpa': 10.0, 'choices': ['civil', 'engphys', 'software']}, {'macid': 'bob2', 'fname': 'bob',
     'lname': 'jill', 'gender':
'male', 'gpa': 9.0, 'choices': ['civil', 'mechanical', 'software']}, {'macid': 'rob1', 'fname': 'rob',
     'lname': 'jill', 'gender':
'male', 'gpa': 7.0, 'choices': ['civil', 'mechanical', 'software']}], 'chemical': [], 'electrical':
     [], 'mechanical': [], 'software': [], 'materials': [], 'engphys': []}

output19 = CalcModule.allocate(Allocate_test4, free_choice4, dept_cap4)

#print(output19)

if output19 == expected_output19:
        print("Test case 4 for allocate(S, F, C) passed")
else:
        print("Test case 4 for allocate(S, F, C) failed")
```

I Code for Partner's CalcModule.py

```
## @file CalcModule.py
#  @author Rohit Saily
#  @brief Module used to process student and department data.
#  @date 2019-01-18

#from a1_constants import MIN_GPA, MAX_GPA, MIN_PASSING_GPA, GENDERS
#from a1_utility import remove_duplicates

""" Helper Functions """
## @brief Allocates a student to their topmost available choice if they have a passing gpa, defined in
    a1_constants.py.
#  @details If a student cannot be allocated to any of their choices, they simply are not allocated.
    If their choice is not found in allocations or spots_available, it is ignored.
#  @param student The student to be allocated to a department, represented by a dictionary that maps
    data categories to data.
#  @param allocations A dictionary that maps a department to a list of student dictionaries already
    allocated.
#  @param spots_available A dictionary that maps a department to the number of students that can be
    allocated to it.
def allocate_topmost_available_choice(student, allocations, spots_available):
    if not (4 <= student['gpa'] <= 12):
        return #Do not allocate the student, they are not passing or have an invalid GPA
    for choice in student['choices']:
        if 0 < spots_available[choice]:
            try:
                allocations[choice].append(student)
            except KeyError:
                continue #The choice is not allocateable, continue to try the next one
            else:
                try:
                    spots_available[choice] -= 1
                except KeyError:
                    allocations[choice].remove(student) #We cannot gurantee the department exists with
                        space, so we cannot allocate the student without it being in the
                        spots_available dictionary
                    continue
                else:
                    return

""" API """
## @brief Sorts students by descending GPA.
#  @details
#      Each student is represented by a dictionary that maps data categories to data.
#      This function does not mutate the original list.
#      Sorts using Tim Sort via Python's sorted function.
#  @param S The list of students represented by dictionaries that map data categories to corresponding
    data.
#  @return The list of dictionaries representing students organized by descending GPA.
def sort(S):
    if S == []:
        return S
    return sorted(S, key=lambda student: student['gpa'], reverse=True)

## @brief Averages the GPA of students, filtered by gender.
#  @details Any student found to have a gpa below the minimum or above the maximum, each defined in
    a1_constants.py, will be ignored in the computation.
#  @param L The list of students, each represented by dictionaries that map data categories to data.
#  @param g The gender to filter by, case insensitive.
#  @return None if no students were found to average otherwise it is the average computed.
def average(L, g):
    if not L:
        return None
    gpas = []
    for student in L:
        if not (4 <= student['gpa'] <= 12):
            continue
        try:
            if student['gender'] == g.lower():
                gpas.append(student['gpa'])
        except KeyError:
            continue
        else:
            pass
    try:
        average = sum(gpas) / len(gpas)
    except ZeroDivisionError:
        return None #There where no values to average hence there is no average
    else:
        return average

## @brief Allocates students to departments based on code defined allocation scheme.
#  @details
#       The code defined allocation scheme is as follows:
#           1. Free choice students are allocated before students without free choice.
#           2. Students with higher GPAs are allocated first.
#           3. Students with failing GPAs or GPAs above the maximum are not allocated.
#           4. Students who cannot fit into any of departments they chose are not allocated to any
     department.
#  @param S The list of students represented by dictionaries that map data categories to corresponding
     data.
#  @param F The list of mac ids of students who have free choice.
#  @param C A dictionary that maps department names to their capacity.
#  @return A dictionary mapping departments to a list of students allocated to that department.
def allocate(S, F, C):
    if not C:
        return {} #No allocations can be made without departments!
    allocations = {}
    spots_available = {department:capacity for department, capacity in C.items()}
    for department in spots_available:
        allocations[department] = []
    if not S:
        return allocations #No students to allocate
    students = []
    students.extend(sort(S))
    #remove_duplicates(students) #To prevent accidentally allocating a student twice if they are
         defined multiple times in the list.
    free_choice_students = []
    for student in students:
        if student['macid'] in F:
            free_choice_students.append(student)
            students.remove(student)
    non_free_choice_students = students #Just to 'rename' the variable and increase code readability
         since students now holds only students without free choice.
    for student in free_choice_students:
        allocate_topmost_available_choice(student, allocations, spots_available)
    for student in non_free_choice_students:
        allocate_topmost_available_choice(student, allocations, spots_available)
    return allocations
```

J Makefile

```
PY = python
PYFLAGS =
DOC = doxygen
DOCFLAGS =
DOCCONFIG = docConfig

SRC = src/testCalc.py

.PHONY: all test doc clean

test:
	$(PY) $(PYFLAGS) $(SRC)

doc:
	$(DOC) $(DOCFLAGS) $(DOCCONFIG)
	cd latex && $(MAKE)

all: test doc

clean:
	rm -rf html
	rm -rf latex
```