Assignment 1 Report

Harsh Patel patelh75

January 25, 2019

The purpose of this report is to give a more in-depth insight of the test cases, note any assumptions or improvements in the program, and comparing our module with the module of a selected partner.

1 Testing of the Original Program

The approach to testing the functionality of my modules was to create a function that would compare the calculated output with the output returned by the code modules. It would display a message indicating if the output was correct or incorrect. Test cases for the modules consist of normal and boundary test cases. The first test case was a normal test case that tested if the sorting function sorted the students based on their GPA in descending order. The second test case was to discover how the program responded to sorting students if they had the same GPA. This was an important test case as it can affect the allocation of students who might have the same GPA and second year program selection. The result was that students appearing near the top of the text file would be placed at the lower index of the sorted list when compared with another student with the same GPA. The third test case was to ensure that a sorted list would stay sorted. The fourth test case tested an empty list to check if the program would return any errors such as not finding the key 'gpa' or not being able to find any lists of dictionaries. This test case did not return any errors, but instead returned an empty list as expected. The next section of test cases tested the function responsible for calculating the average GPA of a specific gender in the student dictionary. The first two test cases returned the average GPA of male or female students depending on gender chosen. The next two test cases were designed to discover the result when no members of a specific gender were in the student list, and the program tested for that specific gender's average GPA. The program terminated with a "ZeroDivisionError" as expected. The function was modified such that a print message would be shown informing the user that no student of this specific gender was present. The final section of test cases tested the function responsible for allocating students to their second year program. The first test case was to test if the function resulted in the correct output when given a list of student dictionaries, department capacities, and a list of free choice students. This function returns a dictionary of departments, each assigned a list of student dictionaries that were allocated respectively. The second test case explored the result of when all free choice students chose a specific program as their first choice, and the number of free choice students assigned were more then the capacity of that specific program. This resulted in all free choice students being allocated to that specific program. Any student without free choice selecting that specific program would not be allocated to their first choice, and would be redirected to their second choice. The last test case tested an empty dictionary as an input to discover any errors that could potentially be presented in a more user-friendly format. This test case returned an empty dictionary.

This assignment required many assumptions dependant on the programmer as it was a natural language specification. The file of containing student information is assumed to be formatted such that each student dictionary is on a new line, the student dictionary is formatted as a python dictionary in the text file, and all information is present and spelled correctly for the keys and values of the student dictionary. The file containing the department capacities is assumed to also be in python dictionary format with departments as keys and capacities as values. Each department dictionary is assumed to be on a new line in the file. The GPA of the students is assumed to be rounded to two decimal places, including when calculating the average GPA for a specific geneder. It is also assumed that free choice students will be allocated to their first choice even if the capacity of that program has been filled. To avoid students not being allocated to any program, it is assumed that the total capacity of all departments is more then the total number of students. Students with a GPA lower than or equal to 4.0 will not be allocated.

2 Results of Testing Partner's Code

The test cases tested with the partner's CalcModule did not run due to an error with the dictionary keys. Some key values defined in the partner's code were different then the key values defined in my code, such as I formatted MacID as "macId" while the partner's code has "macid". Due to these types of error, no test cases were able to run because the program could not find the string "macid" in the text files and dictionaries, and therefore terminated.

3 Discussion of Test Results

3.1 Problems with Original Code

While testing my testCalc module with the partner's CalcModule, I learned that the partner files defined the key values as it was written in the assignment specification. The assignment specification defined MacID as "macid" in the format, and had the first letter of all departments as lowercase. Analyzing these key errors, I was able to depict that my text files and modules specifically follow the key "macId" and had all department names begin with an uppercase, which was not how it was formatted in the assignment specification. Small errors in different variations of how strings were formatted causes the program to return errors and terminate. This is a common issue with strings as it must be specified in a certain format or it will cause errors even if it's the same word. The solution to this issue would simply be to convert all strings to lowercase using the built in python .lower() function enabling the program to run dependant upon spelling rather than string formatting. No matter how the user defined the specific keys, they would all be converted to lowercase. Another problem with my code could arise from the specific formatting of my text files. The module can only read from the files and run the functions if the text files are formatted in the same way as explained above in the report.

3.2 Problems with Partner's Code

After fixing the issues with defining the keys in a specific format using the solution stated above, I was able to run the test cases with the partner's CalcModule. The test cases executed without any errors because the partner's module also relied on the same formatting of the text files as my module. The result of the test cases after fixing the issues was that it passed for some while failing for others. The first test case involved normal sorting, and that test case passed the partner's CalcModule. The next test case involved sorting students with the same GPA. This test case failed as the partner's CalcModule sorts students based on MacID first then GPA. The test cases for calculating the average GPA also fails because the partner's CalcModule rounds the average GPA to three decimal places as opposed to my CalcModule which rounds to two decimal places. Unless the average GPA of the specific gender is calculated exactly to two decimal places or one decimal places, the test case will fail. When testing the test case for when no members of a specific gender are present, the partner's CalcModule simply returns 0.0 while my CalcModule returns a message informing the user that no member from that gender is in the list. For that reason, those test cases also fail for the partner's code. The Allocate function test cases also fail with the partner's code because the program sorts the students based on MacID first, then GPA, so the order of the students in the list corresponding to the department key would not be in the same order. Lists, unlike dictionaries, are not random, so order of the element is an important factor when testing equality. The partner's code would only work if only one student was allocated to each department, and each student had a different first choice. Following that statement, the test case for normal allocation passed, but the test case where every free choice student chose software as their first choice failed. Based on different assumptions made by the programmer throughout this assignment, test cases are expected to fail.

4 Critique of Design Specification

The specification of this assignment required many assumptions to be made by the programmer such as the structure of the files, rounding the gpa of students, allocating students with 4.0 GPA, etc. This was a new approach to programming as many of the decisions relied on the programmer. There were many advantages to working through this kind of assignment. It gives a representation of how problems in the real world are solved. In the software industry a problem is simply presented with no specific specification, and requires the programmer to think of the many approaches and find the most efficient one to implement. Decision making and assumptions is important to solving problems with code as this allows us to develop a thinking to solve many errors, and looking at the program from the perspective of the user. The area that I think needs improvement or needs to added in the assignment is a task that gets the students thinking about how they can make the program more efficient. This could include simply talking about structures of files, removing redundant functions, having less for loops, etc. For example, the entire file for free choice students can be removed and a key of free choice can be added to the student dictionary with options 'Yes' or 'No' as values. This would allow the program to be more efficient as it would have to read from one less file, and would result in removing one entire function. The allocate function would have one less parameter, and less code overall if the free choice selection was included in the main file of students.

5 Answers to Questions

(a) To make the average function more general/flexible, it should have the feature to calculate the average GPA of all members in the program. Along with the string "male" and "female", it should accept the string "All" that would instruct the function to calculate the average GPA of all students. The function would be specified as average(L, "All"). The average function can also have the feature to calculate the average GPA of students allocated in a specific department. A department string with the gender string would instruct the program to calculate the average GPA of male, female, or all students allocated in that program to determine more information such as the cut-off for the year. The "None" string would instruct the function to get the average GPA of all students, not specific only to department. The function would be specified as average(L, "Male", "Software") or average(L, "All", "None") while L represents a list of students which would be used for the allocate function invoked in the average function to get the allocation details. The sort function can be more general by having the option to sort by more parameters such as choosing to sort by "macId", "gpa", "lname", "fname", etc. The function would be specified as sort(S, "macId").

(b) Aliasing is when one variable is assigned to another such that both variables refer to the same object. In this context it would mean that the same list of student dictionaries are assigned to two different variables, and since lists are mutable types, a change to the list assigned to one variable causes the same change to occur in the list assigned to the other variable. Since a dictionary is also a mutable type, aliasing is a problem, as it can cause changes to its keys and values, similar to list elements. A potential strategy to guard against this is to use the python built in copy function. The resulting copyDict = oldDict.copy() would create a copy of the original dictionary while preserving the old one in a different variable. The solution for aliasing lists is to clone it using slice method new = old[:] which will create a new list with the same data.

(c) ReadAllocationData.py could have been tested in testCalc.py if the text files were used for testing in which the functions of CalcModule would require an input created by the text files containing the data. Some isolated test cases to ensure ReadAllocationData functioned correctly could have been simply reading information from the text files given and determining if the returned result was correct. Other boundary test cases would include empty lists, or files of different formats to potentially build stronger code that can solve such problems. CalcModule was selected over ReadAllocationData to be tested because CalcModule uses all of the functions from ReadAllocationData to create inputs for its functions, therefore it would already be tested when testing CalcModule if text files were used to test it. This way an error in ReadAllocationData would be caught and made obvious almost instantly. For this reason, CalcModule would be the logical choice to be tested.

(d) Strings are case sensitive, and this property of strings requires it to be formatted correctly or it will cause an error as seen when testing the partner's CalcModule with my testCalc. A better approach to this would be to use a set and refer to the set element when calculating the average GPA for a specific gender. This would not cause any errors and issues of strings being case sensitive because the strings, "male" and "female", are part of the predefined set male, female, and simply accessed.

(e) Other ways of implementing mathematical tuples in python could be to use classes instead of dictionaries. I would reccomend changing the data structure used in the modules to a student class. A student class would include all the general information for students, and the program would simply have to assign students to this class. Each student would have their own class, and accessors would be used to access the information from the student class which would be much easier then iterating over a list to find the student and the his/her information. Invoking the student class by a specific identifier for each student would result in a more efficient and organized program.

(f) If the data structure inside the custom class changes from a list to a tuple, CalcModule would not need to be modified. The information stored as a tuple are accessed by element as in a list. The only notable difference between a list and tuple is that a tuple is immutable which means that the elements cannot be modified, and tuple itself cannot be modified. Elements cannot be deleted or removed from a tuple, nor can they be appended to a tuple as it uses a fixed memory. Therefore, a tuple behaves similar to a list, but they are immutable, but that does not impact the CalcModule as we are only accessing information stored in the tuple. If the custom class for students uses tuples instead of lists, the CalcModule would still not need to be changed for the same explanation as above.

F Code for ReadAllocationData.py

```
## @file ReadAllocationData.py
#  @author Harsh Patel
#  @brief Reads student information from files.
#  @date 1/15/2019

import ast

## @brief Reads a file and returns a list of dictionaries of student information.
#  @details Function accepts one parameter that is the name of a file containing student information.
#  @param s ("students.txt") name of file being read.
#  @return List of dictionaries of student information.
def readStdnts(s):
    with open(s, "r") as studentFiles:
        studentData = studentFiles.read()
        studentData = studentData.splitlines()
    return [ast.literal_eval(student) for student in studentData]

## @brief Reads a file containing only free choice students and returns their macID's in a list.
#  @param s ("FreeChoice.txt") Name of file being read.
#  @return List of macIds of students with free choice.
def readFreeChoice(s):
    with open(s, "r") as studentChoice:
        studentData = studentChoice.read()
        studentData = studentData.splitlines()
        freeChoiceLst = [ast.literal_eval(stdnt) for stdnt in studentData]
    return [freeStdnt["macId"] for freeStdnt in freeChoiceLst]

## @brief Reads a file containing capacities of each department students will be allocated to and
#         returns the name of the department and capacity in a dictionary.
#  @param s ("Capacity.txt") Name of file being read.
#  @return Dictionary with each department name and capacity
def readDeptCapacity(s):
    capacity = {}
    deptLst = readStdnts(s)
    for dept in deptLst:
        capacity.update(dept)
    return capacity
```

G Code for CalcModule.py

```
## @file CalcModule.py
#  @author Harsh Patel
#  @brief Allocates students to second year programs based on free choice and GPA.
#  @date 1/15/2019

from ReadAllocationData import *
import operator
import ast

## @brief Sorts the list of student dictionaries based on student GPA in descending order.
#  @param S (studentLst) List of student dictionaries created by function readStdnts(s).
#  @return Sorted list of student dictionaries in descending order based on student GPA score.
def sort(S):
    #from source
        https://stackoverflow.com/questions/72899/how-do-i-sort-a-list-of-dictionaries-by-a-value-of-the-dictionary
    sortGpaLst = sorted(S, key=operator.itemgetter('gpa'), reverse=True)
    return sortGpaLst

## @brief Computes average GPA of all males or females depending on the gender selected.
#  @param L (studentLst) List of student dictionaries created by function readStdnts(s).
#  @param g (gender) Gender, male or female, to compute average GPA.
#  @return Average GPA of a specfic gender.
def average(L, g):
    gradeSum = 0
    numGender = 0
    for studnt in L:
        if (studnt["gender"] == g.lower()):
            gradeSum += studnt["gpa"]
            numGender += 1
    if (numGender > 0):
        return round(gradeSum/numGender, 2)
    else:
        return ("No %s students in program" %(g.lower()))

## @brief Allocates students to a program as long as their GPA is above 4.0.
#  @details Allocates students with a GPA above 4.0 to a program beginning
#           from allocating free choice students first, then normal students from
#           highest GPA first. Students will be allocated to their first choice unless its full.
#           If their second choice is full, then they are allocated to their third choice.
#  @param S (studentLst) List of student dictionaries created by function readStdnts(s).
#  @param F (freeChoiceLst) List of macIDs of students with free choice created by readFreeChoice(s).
#  @param C (capacityDict) Dictionary with each department and their corresponding capacities.
#  @return Dctionary with each a list of student dictionaries allocated to each department
def allocate(S, F, C):

    allocateDict = {}

    #Departments
    Software = []
    Civil = []
    Chemical = []
    Electrical = []
    Materials = []
    Engphys = []
    Mechanical = []

    #Copy of list of student dictionaries created by function readStdnts(s)
    tempLst = S

    #Copy of dictionary with each department and corresponding capacities
    capacityDict = C

    #allocate free choice students
    for freeStdnts in F:
        count = -1
        for normStdnts in tempLst:
            count += 1
            if (normStdnts["macId"] == freeStdnts):
                if(normStdnts["gpa"] > 4.0):
                    allocation = vars()[normStdnts["choices"][0]]
                    allocation.append(normStdnts)
                    capacityDict[normStdnts["choices"][0]] -= 1
                    del(tempLst[count])
                else:
                    del(tempLst[count])
```

```
#sort students based on highest GPA first
highToLow = sort(tempLst)

#allocate non-free choice students
for students in highToLow:
    if (students["gpa"] > 4.0):
        progAllo = vars()[students["choices"][0]]
        progAllo1 = vars()[students["choices"][1]]
        progAllo2 = vars()[students["choices"][2]]
        if (capacityDict[students["choices"][0]] > 0):
            progAllo.append(students)
            capacityDict[students["choices"][0]] -= 1
        elif (capacityDict[students["choices"][1]] > 0):
            progAllo1.append(students)
            capacityDict[students["choices"][1]] -= 1
        elif (capacityDict[students["choices"][2]] > 0):
            progAllo2.append(students)
            capacityDict[students["choices"][2]] -= 1

departments = C.keys()

for program in departments:
    allocateDict[program] = vars()[program]

return allocateDict
```

H Code for testCalc.py

```
from ReadAllocationData import *
from CalcModule import *
import operator
import ast

def testEqual(test, result, testName):
    if (test == result):
        print("Test Description : %s\n" %(testName))
        print("%s == %s\n" % (test, result))
        print("TEST PASSED!\n")
    else:
        print("Test Description : %s\n" %(testName))
        print("%s != %s\n" % (test, result))
        print("TEST FAILED\n")

#Normal test case to see if List is sorted correctly
def testSort1():
    stdntFile = readStdnts("sortGpaStdnts.txt")
    sortedLst = sort(stdntFile)
    testEqual(sortedLst, [{'macId' : 'patelh75', 'fname' : 'Harsh', 'lname' : 'Patel', 'gender' :
        'male', 'gpa' : 10.6, 'choices' : ['Software', 'Mechanical', 'Engphys']},
                          {'macId' : 'cody9', 'fname' : 'Cody', 'lname' : 'Codes', 'gender' : 'male',
                              'gpa' : 9.2, 'choices' : ['Mechanical', 'Engphys', 'Chemical']},
                          {'macId' : 'brain8', 'fname' : 'Shawn', 'lname' : 'Brain', 'gender' :
                              'male', 'gpa' : 8.5, 'choices' : ['Software', 'Chemical',
                              'Materials']}],
            "Sorting File GPA descending")


#Test how order of sorting occurs when some students have the same GPA
def testSort2():
    stdntFile = readStdnts("sortSameGpa.txt")
    sortedFile = sort(stdntFile)
    testEqual(sortedFile, [{'macId' : 'patelh75', 'fname' : 'Harsh', 'lname' : 'Patel', 'gender' :
        'male', 'gpa' : 10.5, 'choices' : ['Software', 'Mechanical', 'Engphys']},
                           {'macId' : 'cody9', 'fname' : 'Cody', 'lname' : 'Codes', 'gender' : 'male',
                               'gpa' : 10.5, 'choices' : ['Mechanical', 'Engphys', 'Chemical']},
                           {'macId' : 'brain8', 'fname' : 'Shawn', 'lname' : 'Brain', 'gender' :
                               'male', 'gpa' : 8.5, 'choices' : ['Software', 'Chemical',
                               'Materials']},
                           {'macId' : 'grande4', 'fname' : 'Ariana', 'lname' : 'Grande', 'gender' :
                               'female', 'gpa' : 8.5, 'choices' : ['Materials', 'Chemical',
                               'Engphys']}],
             "Students with Same GPA : Students entered first in the file appear before other student
                 with same GPA in Sorted List")

#Test to see sorted list stays sorted
def testSort3():
    stdntFile = readStdnts("AlreadySorted.txt")
    sortedFile = sort(stdntFile)
    testEqual(sortedFile, [{'macId' : 'patelh75', 'fname' : 'Harsh', 'lname' : 'Patel', 'gender' :
        'male', 'gpa' : 10.5, 'choices' : ['Software', 'Mechanical', 'Engphys']},
                           {'macId' : 'cody9', 'fname' : 'Cody', 'lname' : 'Codes', 'gender' : 'male',
                               'gpa' : 9.2, 'choices' : ['Mechanical', 'Engphys', 'Chemical']},
                           {'macId' : 'brain8', 'fname' : 'Shawn', 'lname' : 'Brain', 'gender' :
                               'male', 'gpa' : 8.5, 'choices' : ['Software', 'Chemical',
                               'Materials']}],
             "Sorted File remains Sorted in List")

#Test if an empty file will cause any errors or simply return an empty list
def testSort4():
    stdntFile = readStdnts("empty.txt")
    sortedFile = sort(stdntFile)
    testEqual(sortedFile, [], "Empty File returns empty list")

#Normal Test case to calculate average Male GPA
def testAverageMale1():
    stdntFile = readStdnts("students.txt")
    maleAverage = average(stdntFile, "male")
    testEqual(8.22, maleAverage, "Average GPA of male students")

#Normal Test case to calculate average female GPA
def testAverageFemale1():
    stdntFile = readStdnts("students.txt")
    femaleAverage = average(stdntFile, "female")
    testEqual(9.1, femaleAverage, "Average GPA of female students")
```

```python
#Test Case when no male students are enrolled and user wants average male GPA
def testAverageMale2():
    stdntFile = readStdnts("onlyFemales.txt")
    maleAverage = average(stdntFile, "male")
    testEqual("No male students in program", maleAverage, "Calculating average GPA of male students
        when no male students in program")


#Test Case when no female students are enrolled and user wants average female GPA
def testAverageFemale2():
    stdntFile = readStdnts("onlyMales.txt")
    femaleAverage = average(stdntFile, "female")
    testEqual("No female students in program", femaleAverage, "Calculating average GPA of female
        students when no female students in program")

#Test case for normal allocation of students into selected programs
def testAllocate1():
    stdntFile = readStdnts("students.txt")
    freeChoice = readFreeChoice("freeChoice.txt")
    capacityFile = readDeptCapacity("depCapacity.txt")
    testEqual({"Software" : [{'macId' : 'patelh75', 'fname' : 'Harsh', 'lname' : 'Patel', 'gender' :
        'male', 'gpa' : 10.6, 'choices' : ['Software', 'Mechanical', 'Engphys']},
                             {'macId' : 'brain8', 'fname' : 'Shawn', 'lname' : 'Brain', 'gender' :
                                 'male', 'gpa' : 8.5, 'choices' : ['Software', 'Chemical',
                                 'Materials']},
                             {'macId' : 'Jacky3', 'fname' : 'Jack', 'lname' : 'Jacky', 'gender' :
                                 'male', 'gpa' : 7.2, 'choices' : ['Software', 'Electrical',
                                 'Engphys']}],
              "Electrical" : [{'macId' : 'kenway1', 'fname' : 'Edward', 'lname' : 'Kenway', 'gender'
                  : 'male', 'gpa' : 9.8, 'choices' : ['Electrical', 'Engphys', 'Chemical']}],
              "Materials" : [{'macId' : 'grande4', 'fname' : 'Ariana', 'lname' : 'Grande', 'gender' :
                  'female', 'gpa' : 9.5, 'choices' : ['Materials', 'Chemical', 'Engphys']}],
              "Engphys" : [{'macId' : 'pam4', 'fname' : 'Pammy', 'lname' : 'Pam', 'gender' :
                  'female', 'gpa' : 8.2, 'choices' : ['Engphys', 'Electrical', 'Civil']}],
              "Chemical" : [{'macId' : 'jen2', 'fname' : 'Jenny', 'lname' : 'Jen', 'gender' :
                  'female', 'gpa' : 11.2, 'choices' : ['Chemical', 'Civil', 'Engphys']}],
              "Mechanical" : [{'macId' : 'cody9', 'fname' : 'Cody', 'lname' : 'Codes', 'gender' :
                  'male', 'gpa' : 9.2, 'choices' : ['Mechanical', 'Engphys', 'Chemical']}],
              "Civil" : [{'macId' : 'jessie1', 'fname' : 'Jes', 'lname' : 'Jessie', 'gender' :
                  'female', 'gpa' : 7.5, 'choices' : ['Civil', 'Software', 'Materials']}]},
             allocate(stdntFile, freeChoice, capacityFile), "Normal Student Allocation")


#more free choice students then number of seats of popular program so they should be allocated even if
    seats are full
#non-free choice students selecting that filled program will be allocated to their second choice
def testAllocate2():
    stdntFile = readStdnts("studentTestAllocate.txt")
    freeChoice = readFreeChoice("freeChoiceAllSoftware.txt")
    capacityFile = readDeptCapacity("depCapacity.txt")
    testEqual({"Software" : [{'macId' : 'patelh75', 'fname' : 'Harsh', 'lname' : 'Patel', 'gender' :
        'male', 'gpa' : 10.6, 'choices' : ['Software', 'Mechanical', 'Engphys']},
                             {'macId' : 'brain8', 'fname' : 'Shawn', 'lname' : 'Brain', 'gender' :
                                 'male', 'gpa' : 8.5, 'choices' : ['Software', 'Chemical',
                                 'Materials']},
                             {'macId' : 'cody9', 'fname' : 'Cody', 'lname' : 'Codes', 'gender' :
                                 'male', 'gpa' : 9.2, 'choices' : ['Software', 'Engphys',
                                 'Chemical']},
                             {'macId' : 'jen2', 'fname' : 'Jenny', 'lname' : 'Jen', 'gender' :
                                 'female', 'gpa' : 11.2, 'choices' : ['Software', 'Civil',
                                 'Engphys']},
                             {'macId' : 'jessie1', 'fname' : 'Jes', 'lname' : 'Jessie', 'gender' :
                                 'female', 'gpa' : 7.5, 'choices' : ['Software', 'Engphys',
                                 'Materials']}],
              "Electrical" : [{'macId' : 'kenway1', 'fname' : 'Edward', 'lname' : 'Kenway', 'gender'
                  : 'male', 'gpa' : 9.8, 'choices' : ['Electrical', 'Engphys', 'Chemical']},
                               {'macId' : 'Jacky3', 'fname' : 'Jack', 'lname' : 'Jacky', 'gender' :
                                   'male', 'gpa' : 7.2, 'choices' : ['Software', 'Electrical',
                                   'Engphys']}],
              "Materials" : [{'macId' : 'grande4', 'fname' : 'Ariana', 'lname' : 'Grande', 'gender' :
                  'female', 'gpa' : 9.5, 'choices' : ['Materials', 'Chemical', 'Engphys']}],
              "Engphys" : [{'macId' : 'pam4', 'fname' : 'Pammy', 'lname' : 'Pam', 'gender' :
                  'female', 'gpa' : 8.2, 'choices' : ['Engphys', 'Electrical', 'Civil']}],
              "Chemical" : [],
              "Mechanical" : [],
              "Civil" : []},
             allocate(stdntFile, freeChoice, capacityFile), "More free choice students enrolled in a
                 certain department then capacity of program")
```

```
#Test case to see if empty list or file raises any errors
def testAllocate3():
    stdntFile = readStdnts("empty.txt")
    freeChoice = readFreeChoice("empty.txt")
    capacityFile = readDeptCapacity("depCapacity.txt")
    testEqual({"Software" : [], "Electrical" : [], "Materials" : [],
               "Engphys" : [], "Chemical" : [], "Mechanical" : [], "Civil" : []},
              allocate(stdntFile, freeChoice, capacityFile), "Empty file returns Empty dictionary with
                  department names and empty lists")


def test():
    testSort1()
    testSort2()
    testSort3()
    testSort4()
    testAverageMale1()
    testAverageFemale1()
    testAverageMale2()
    testAverageFemale2()
    testAllocate1()
    testAllocate2()
    testAllocate3()
test()
```

I Code for Partner's CalcModule.py

```
## @file CalcModule.py
#  @author Justin Rosner
#  @brief Performs calculations/places students in departments
#  @date 1/17/2019

from operator import itemgetter

from random import choice

from copy import deepcopy

from ReadAllocationData import *


## @brief This function sorts the list of students according to gpa
#  @param s List of the dictionaries, not yet sorted
#  @return List of dictionaries that is now sorted
def sort(S):
    # The assumption is that if two people have the same gpa,
    # they will be sorted in alphabetical order according to macid
    macidSort = sorted(S, key=itemgetter('macId'))
    sortedList = sorted(macidSort, key=itemgetter('gpa'), reverse=True)
    return (sortedList)


## @brief This function gets the average gpa of the desired gender
#  @param L List of dictionaries of students
#  @param g String representing the gender of a student
#  @return List of the average gpa of the desired gender, If there
#  is an error in reading the file or the gender entry is invalid it
#  will return a value of 0.0
def average(L, g):
    total = 0.0
    count = 0.0

    # Making sure that the gender entry is valid
    if not(g == 'male' or g == 'female'):
        print("Invalid entry, please enter either 'male' or 'female'")

    else:
        for student in L:
            if (student['gender'] == g):
                total += student['gpa']
                count += 1.0

    # If count is still 0 at this point it means that there was an error in
    # reading the file, or that the file is empty
    if (count == 0.0):
        return 0.0
    else:
        # Assuming that the average will be rounded to 3 decimals
        average = round((total / count), 3)
        return (average)


## @brief This function allocates first years into upper year programs
#  @param S List of dictionaries of students, unsorted
#  @param F List of students with free choice
#  @param C Dictionary of department capacities
#  @return Dictionary of the form {'department':[list of students]}
def allocate(S, F, C):

    finalDepartments = {'Civil': [], 'Chemical': [], 'Electrical': [],
                        'Mechanical': [], 'Software': [], 'Materials': [],
                        'Engphys': []}
    sortedList = sort(S)
    freeList = deepcopy(C)

    # Allocating free choice students first
    for freeStudent in sortedList:
        # Here I'm assuming that if a student has a GPA <= 4.0 they will not
        # be allocated into any second year program even with freechoice
        if ((freeStudent['gpa'] <= 4.0) or (freeStudent['macId'] not in F)):
            continue

        # Here I am assuming that students that have free choice will get
```

```
    # their top choice no matter what
    else:
        finalDepartments[freeStudent['choices'][0]].append(freeStudent)
        freeList[freeStudent['choices'][0]] -= 1

# Now allocating the other students according to GPA
for student in sortedList:

    # This checks the number of spots remaining in all level 2 programs
    numSpots = 0
    for department in freeList:
        numSpots += freeList[department]

    if ((student['gpa'] <= 4.0) or (student['macId'] in F)):
        continue

    else:
        if (freeList[student['choices'][0]] > 0):
            finalDepartments[student['choices'][0]].append(student)
            freeList[student['choices'][0]] -= 1

        elif (freeList[student['choices'][1]] > 0):
            finalDepartments[student['choices'][1]].append(student)
            freeList[student['choices'][1]] -= 1

        elif (freeList[student['choices'][1]] > 0):
            finalDepartments[student['choices'][1]].append(student)
            freeList[student['choices'][1]] -= 1

        # Here I am assuming that if all the departments are full then
        # it means that the student did not make it to level 2
        elif(numSpots == 0):
            continue

        # Here I'm assuming that if all three of the students' choices are
        # full then they will randomly be placed in an open department
        else:
            randDept = choice(list(freeList))
            while (freeList[randDept] == 0):
                randDept = choice(list(freeList))
            finalDepartments[randDept].append(student)

print(finalDepartments)
return (finalDepartments)
```

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
```