Assignment 1 Solution

Donisius Wigie - wigied

January 25, 2019

The modules ReadAllocationData.py and CalcModule.py are used to allocate first year engineering students into their second year programs. ReadAllocationData.py will read relevant data from files and put them into a form specified by the assignment. CalcModule.py performs operations on the data and allocates the first year students into their second year programs.

1 Testing of the Original Program

My approach towards coming up with the testing module was to think of ways in order to automate the module as much as possible. I wanted the test cases to make as little assumptions as possible about the code. This involves not having to change the format of the files, and providing input by creating data by itself.

I created a function to test the sort function and see if it actually sorted the gpas of the students in descending order.

I also created a function failures(s) to obtain a list of macIDs who have a gpa that is less than 4.0 using the list of dictionaries produced by readStdnts.

The belongOrNot(A, F) function is to determine if any of the students who have a gpa below 4.0 were allocated into a second year program. It takes the list of students who failed produced by failures(s) and the dictionary of allocations produced by allocate(S, F, C).

The firstChoices(F, S) function puts the macIDs of the first choice students along with their first choice into a list so it is convenient for the fcAllocated(A, fc) function to see if everyone who had first choice got their first choice pick.

These functions are used to test all regular inputs. The first four test cases were just regular input tests to see if the CalcModule functions will work for inputs which should be obvious.

For boundary cases, I tried to think of as many as I could which are possible inputs a user could provide the programs. The first boundary case is to see if the function can

handle an empty department capacity file. The first boundary case forced me to think of a way to deal with an empty department list. In the beginning of my allocate function I added a feature which makes sure the department capacity file is not empty or missing any of the seven streams. It also tests to see if it could deal with an empty student file. Another boundary case regarding department list tests to see if allocate can deal with greater number of students who have a gpa of 4.0 or over than there are department seats. I needed to let the user know that there is not enough space or not all students are able to be allocated. I added a feature in my allocate function that checks to see that there is enough department space for the students.

The third boundary case is to test the average function by entering a non-male or non-female gender as the gender input. Ideally the function would raise an exception.

The fourth boundary case is to test if allocate can handle students with a negative gpa. This was not an issue in my program as it checks if the gpa is below 4.0 before allocating.The fifth boundary case which tests if allocate can handle students with a gpa above 12 properly raised issues for me. I fixed this by checking to see if gpa is below or equal to 12 in addition to checking if gpa is above or equal to 4.0 before allocating.

The sixth boundary case is to see if the average function can handle an empty list. This raised a problem in my average function since it would cause my program to try to divide by 0. I fixed this by checking to see if the list is empty and terminating before being able to carry out the rest of the operations.

I also added tests to see if allocate can handle nonsensical inputs which I dealt with by adding a feature to see if the S-input is of type list, if the F-input is of type list and if the C-input is of type dictionary.

My allocate function has a feature which deals with the unlikely chance that an overwhelming amount of people choose the same three top choices and some do not get allocated into any department. In that case, my allocate function will let the user know that 'macID' has not been allocated into any function and they will be put into a random department temporarily until the issue is resolved. I could not think of a way to automate this testing since people may deal with this is a multitude of ways, so I will test the one case manually and see if the program deals with it in a fair way.

2 Results of Testing Partner's Code

The outputs for my test driver used on my partners code:

```
REGULAR INPUTS:
********************************************************************************
The list is sorted properly - PASSED
Students with below a 4.0 gpa:  ['daffyd', 'daffyd2', 'daffyd2']
Students who have below 4.0 were not allocated - PASSED
Regular inputs for allocate - PASSED
Inputs for average function: Regular inputs, ie. supplied files/binary genders
Average function test - PASSED
First choice students:  [['wigied', 'software'], ['kev42', 'materials'], ['frsdf8', 'software'], [
igied2', 'chemical'], ['frsdf82', 'software'], ['jacoba2', 'electrical'], ['chenge2', 'civil'], ['
fcAllocated test failed
BOUNDARY CASES:
********************************************************************************
Input for Empty department test:  {} expected to not crash
there was an exception  'materials'
Input for Empty Student List Test:  [] expected to not crash
Empty Student List Test - PASSED
Input for gender - average function:  apache attack helicopter expected to not crash
Not A Real Gender Test - PASSED
input for allocation function:  This is some random nonsense expected to not crash
there was an exception  string indices must be integers
Negative gpa test:
Input for studentList =  [{'macid': 'wigied', 'fname': 'Donisius', 'lname': 'Wigie', 'gender': 'ma
there was an exception  'software'
Excessively high gpa test:
Input for studentList =  [{'macid': 'wigied', 'fname': 'Donisius', 'lname': 'Wigie', 'gender': 'ma
there was an exception  'software'
Empty list parameter for average function:
Input for average function:  []
there was an exception  division by zero
********************************************************************************
The total score is:  5 out of 11 tests passed
```

After putting my partner's code through my test driver, it passed 5 out of 11 of the tests. The code was able to pass the regular inputs except for the allocation of all free choice students. My partner's code was able to pass 2/7 of the boundary case tests. It passed the empty student list test for the allocate function and the not a real gender input for the average function. For the allocate function, the code failed the empty department test, the random nonsense inputs test, the negative gpa test and the excessively high gpa test. For the average function, it failed the empty list test.

3 Discussion of Test Results

While creating the testCalc.py module, it brought up a lot of issues with my code that I would not have thought about if I was not actively looking for ways my program could be broken. I mentioned earlier that I went into the creation of my test module with the idea of trying to make a little assumptions possible. After putting my partner's code through my test driver, the test results made me realize that some of my test cases make too many assumptions. The best example is the "nonsense" inputs where the test driver would generate an input for the functions which do not have the right input type. The reason I do not think this is a good test is because when someone abuses the program like that by providing input which makes no sense, you can sometimes argue that it may be better just to allow the program to crash. I made the assumption that the program should catch the exception and that is why the test gives me points for that. Someone else could assume that it should just allow it to crash and that is why the test does not give them points. As mentioned earlier, in the event that an overwhelming majority of people choose the same three top choices and some students are unable to be allocated into any of their top choices, then it would just place them into temporary spots. I needed to do this test manually, but looking at my partner's code they also made this same assumption.

3.1 Problems with Original Code

In my discussion of the test results, I commented that in the process of creating my testCalc.py module, it brought up a lot of issues with my code. The most obvious issue was the empty student list for the average function. I did not consider an empty student list as a possible output and because of this I did not realize that an empty list would cause a division by zero to occur. From this I created multiple test cases for my allocate function to deal with empty input and the empty dictionary (empty department) caused issues. I fixed this by checking if it is empty and if there is enough total seats across all departments for all the students. I also did not consider how my program would deal with a gpa greater than 12.0, which I fixed by not allocating anyone who has a gpa less than 4.0 or greater than 12.0.

3.2 Problems with Partner's Code

My test driver brought up some problems with my partner's code, many of which my code had before I came up with the test cases. The first test that failed showed that not everyone who had free choice did not get their first choice department. My functions to test this determined someone did not get allocated and when I manually checked some free choice students did not get put into their first choice. The second test that failed

showed that the allocate function is not able to deal with an empty department list or when there are more students with a passing gpa than there are seats for them to be allocated into. The third test that failed showed that it isn't able to catch the nonsense input exceptions which as mentioned in my discussion of the test results is not entirely wrong. The fourth and fifth tests that failed showed that the program is not able to deal with incorrect grade inputs, i.e, negative gpa or a gpa above 12.0. The sixth test that failed showed that the average function is not able to take an empty list otherwise a division by zero would occur.

4 Critique of Design Specification

The specification of this assignment imposed design decisions which in my opinion could be improved upon. For example, a better implementation for storing the information of the students would be to store the information in a student class for each student. It would be a lot easier to change and access information regarding each student if all the information is stored in a student class. I think keeping the names of students with free choice in a list was a simple but effective design specification since it is easy to access. Another way is to also keep this information in the student class mentioned earlier. This might make it slightly harder to quickly access all free choice names, and slightly easier to know if each individual student has free choice. I believe keeping the information in a list was still a good design decision. As for department capacities I can not think of a better way of implementing that.

5 Answers to Questions

(a) average(L, g) can be made more general by being able to get the average of more than exclusively male or exclusively female. For example a more flexible average function would also be able to take another input (eg. 'all') and take the average of all the students present in the student list input. Another feature that would make the average function more flexible can be to calculate the anticipated cut off average for the departments based off the grades of the students, the demand for each department, and the actual amount of space available in each department. Such a feature would be a difficult task to implement but would make the function extremely versatile. Along a similar line of thinking, sort(s) could be made more general by being able to sort in both ascending order or descending order depending on what the user wants.

(b) We were allowed to make the assumption that aliasing will not occur in the dictionaries in our lists which in this context means that making changes in one dictionary will

affect the others. In general, aliasing is a concern with dictionaries because dictionaries in python are mutable. If you create a variable to take the value of a dictionary and change the variable in any way, the original dictionary will also change. A way that you may guard against this is to create a new instance of the same dictionary instead of making them all connected to the same original dictionary. I implemented such a feature in my readStsnts function.

```
for i in range(numberOfLines - 1):
    stdntInfo = {'macid': None, 'fname': None, 'lname': None,
                 'gender': None, 'gpa': None, 'choices': None }
    temp = file.readline()
    temp2 = temp.split(" ")
    choices = [temp2[5], temp2[6], temp2[7].replace("\n", "").replace("\r", "")]
    stdntInfo['macid'] = temp2[0]
    stdntInfo['fname'] = temp2[1]
    stdntInfo['lname'] = temp2[2]
    stdntInfo['gender'] = temp2[3]
    stdntInfo['gpa'] = float(temp2[4])
    stdntInfo['choices'] = choices
    dictList.append(stdntInfo)
```

In readStdnts, I instantiate a different dictionary by putting a new dictionary inside the for loop. This is done in order to create a different dictionary for each student and to avoid the problems that dictionaries have with aliasing.

(c) The ReadAllocationData.py module is very difficult to test since the way the data is stored in the files is open-ended and because of this it could vary greatly in how people implement this module. For example, you may implement the data files as a csv and use functions to parse the csv files. Or you can just have a regular text file and have the data be organized in any way you want. Because it is so open ended, it is hard to come up with test cases for ReadAllocationData.py and that is why CalcModule.py was selected as the one to be tested. Some test cases to test ReadAllocationData.py can be to see if the functions put the data into the correct form, i.e, does the readStdnts function create a list of dictionaries containing the students' macIDs, lastnames, firstnames, gpas, and program choices? Is the free choice in list form? Is the department capacities in a dictionary?

(d) One problem of using strings as keys in several dictionaries and to represent members of finite sets is that it can be prone to typos. It is very convenient to use strings as representations of members of finite sets. For dictionaries in particular , it could cause many strange issues. Since there is very little protection against misspelled field names and field names can be removed or added easily at any time in a dictionary, it can cause the program to add or remove fields unintentionally. A better approach would be to use namedtuple. By using a namedtuple, it solves this issue by ensuring

that the correct field names are used. In addition, namedtuples are also more memory efficient than regular classes.

(e) The mathematical notion of a tuple is a finite ordered sequence of elements. Other options for implementing this in python is by writing a custom class. Classes allow you to define data structures which may be reused and provide the same set of fields with every instantiation. Similar to a structure in C, a python custom class may be created in order to mimic a data structure which stores various information. The way that the information would be stored is by providing the data through a tuple and creating a new instance of the class. I would definitely recommend changing the data structure used in the code modules to use this custom class, as it is much neater and less error prone. I would create the custom student class as follows:

```
Class Student:
    def __init__(self, macid, fname, lname, gender, gpa, choices):
        self.macid = macid
        self.fname = fname
        self.lname = lname
        self.gender = gender
        self.gpa = gpa
        self.choices = choices
```

(f) If the list of strings was changed to a different data structure like a tuple, then the CalcModule.py wouldn't need to be modified since although a tuple can not be modified, the information is still able to be accessed. In CalcModule.py, there is no need to modify the list of strings for the student's choices, only access to the information is needed. If a custom class was written for students and CalcModule was modified accordingly, then CalcModule would still need to be modified further. The way that my program goes through the student information and checks the choices to allocate the students accordingly is to parse through the information using a for loop. It loops through the top 3 choices and checks if there is room in the first choice department and if not move on to the second choice and if not then move on to the third choice. In order to use the methods from the custom class, I would need to change the implementation to index the choices of the students using the function and not a loop, and to check for true if there is no other choices. This implementation is a lot more flexible as it is able to adapt to an increase in the number of top choices a student can have. Students can also have a different number of top choices.

F Code for ReadAllocationData.py

```
## @file readAllocationData.py
#  @author Donisius Wigie
#  @brief A module to perform calculations on a list of students
#  @date 2019-01-12

## @brief Returns a list of dictionaries of student information
#  @details Each dictionary in the list corresponds to one student
#  @param s Corresponds to a filename containing student information
def readStdnts(s):
    dictList = []
    file = open(s, "r")
    file.seek(0)
    numberOfLines = sum(1 for line in file)
    file.seek(0)
    for i in range(numberOfLines - 1):
        stdntInfo = {'macid': None, 'fname': None, 'lname': None,
                     'gender': None, 'gpa': None, 'choices': None }
        temp = file.readline()
        temp2 = temp.split(" ")
        choices = [temp2[5], temp2[6], temp2[7].replace("\n", "").replace("\r", "")]
        stdntInfo['macid'] = temp2[0]
        stdntInfo['fname'] = temp2[1]
        stdntInfo['lname'] = temp2[2]
        stdntInfo['gender'] = temp2[3]
        stdntInfo['gpa'] = float(temp2[4])
        stdntInfo['choices'] = choices
        dictList.append(stdntInfo)
    file.close()
    return dictList

## @brief Returns a list of strings, where each entry in the list corresponds to the macid of a
    student with free choices
#  @param s Corresponds to a filename containing the macids of students who have free choice
def readFreeChoice(s):
    file = open(s, "r")
    temp = file.readline()
    freeChoice = temp.split(" ")
    file.close()
    return freeChoice

## @brief Returns a dictionary where the key value pairs are {'dept': integer}
#  @details The dictionary contains the capacity of each Department
#  @param s Corresponds to a filename containing the capacities for each department
def readDeptCapacity(s):
    deptCap = {}
    file = open(s, "r")
    numberOfLines = sum(1 for line in file)
    file.seek(0)
    for i in range(numberOfLines):
        temp = file.readline()
        temp2 = temp.split(" ")
        deptCap[temp2[0]] = int(temp2[1].replace("\n", ""))
    file.close()
    return deptCap
```

G Code for CalcModule.py

```
## @file CalcModule.py
#  @author Donisius Wigie
#  @brief A module to perform calculations on a list of students
#  @date 2019-01-12

from ReadAllocationData import *

## @brief Returns a list of student dictionaries in sorted order
#  @details Sorts students from highest to lowest gpa
#  @param S A list of the dictionaries created by function readStdnt(s)
def sort(S):
    for i in range(S.__len__()):
        for j in range(i + 1, S.__len__()):
            if(S[i]['gpa'] < S[j]['gpa']):
                S[i], S[j] = S[j], S[i]
    return S

## @brief Returns the average gpa of male or female students
#  @param L A list of the dictionaries created by function readStdnt(s)
#  @param g A string representing the gender (male or female)
def average(L, g):
    possible = ['male', 'female']
    if(g not in possible):
        return 4
    if(not L):
        return 5
    sum = 0
    total = 0;
    for i in range(L.__len__()):
        if(L[i]['gender'] == g):
            sum += L[i]['gpa']
            total += 1
    return sum/total

# test = average([], 'male')
# print(test)

## @brief Returns a dictionary with the department and list of students which have been allocated into
     each program
#  @details Students who are not allocated to any of their top 3 picks due to no space in those
     departments will be temporarily placed in a department with space
#  @param S A list of the dictionaries created by function readStdnt(s)
#  @param F A list of students with free choice created by function readFreeChoice(s)
#  @param C A dictionary of department capacities created by readDeptCapacity(s)
def allocate(S, F, C):
    stdntAllocations = {'civil': None, 'chemical': None, 'electrical': None,
                        'mechanical': None, 'software': None, 'materials': None,
                        'engphys': None}
    #Checks to make sure inputs are not nonsense
    if(not isinstance(S, list) or not isinstance(F, list) or not isinstance(C, dict)):
        print("Make sure student info and free choice is in list format and department capacity is in
             a dictionary")
        return
    totalSpace = sum(C.values())
    if(totalSpace < S.__len__()):
        print("Departments dont have enough space!")
        return
    if('civil' not in C or C['civil'] < 1):
        print('Department Capacity is missing "civil"' or C['civil'] < 1)
        return
    if('chemical' not in C or C['chemical'] < 1):
        print('Department Capacity is missing "chemical" or has less than 1 spot')
        return
    if('electrical' not in C or C['electrical'] < 1):
        print('Department Capacity is missing "electrical" or has less than 1 spot')
        return
    if('mechanical' not in C or C['mechanical'] < 1):
        print('Department Capacity is missing "mechanical" or has less than 1 spot')
        return
    if('software' not in C or C['software'] < 1):
        print('Department Capacity is missing "software" or has less than 1 spot')
        return
    if('materials' not in C or C['materials'] < 1):
        print('Department Capacity is missing "materials" or has less than 1 spot')
        return
    if('engphys' not in C or C['engphys'] < 1):
```

```
    print('Department Capacity is missing "engphys" or has less than 1 spot')
    return

civil = []
chemical = []
electrical = []
mechanical = []
software = []
materials = []
engphys = []

#allocate all first choice students if gpa is above 4.0
for i in range(F.__len__()):
    for j in range(S.__len__()):
        if(F[i] == S[j]['macid'] and S[j]['gpa'] >= 4.0):
            if(S[j]['choices'][0] == 'civil'):
                civil.append(S[j]['macid'])
            if(S[j]['choices'][0] == 'chemical'):
                chemical.append(S[j]['macid'])
            if(S[j]['choices'][0] == 'electrical'):
                electrical.append(S[j]['macid'])
            if(S[j]['choices'][0] == 'mechanical'):
                mechanical.append(S[j]['macid'])
            if(S[j]['choices'][0] == 'software'):
                software.append(S[j]['macid'])
            if(S[j]['choices'][0] == 'materials'):
                materials.append(S[j]['macid'])
            if(S[j]['choices'][0] == 'engphys'):
                engphys.append(S[j]['macid'])

#sort the list in descending order based on gpa
#then begin allocating the rest of the students
#will create space for third choice if students dont get into any program but are above the cutoff
sort(S)
for j in range(S.__len__()):
    if(S[j]['macid'] not in F):
        for i in range(3):
            if(S[j]['choices'][i] == 'civil' and civil.__len__() < C['civil'] and S[j]['gpa'] >=
                4.0 and S[j]['gpa'] < 13):
                civil.append(S[j]['macid'])
                break;
            if(S[j]['choices'][i] == 'chemical' and chemical.__len__() < C['chemical'] and
                S[j]['gpa'] >= 4.0 and S[j]['gpa'] < 13):
                chemical.append(S[j]['macid'])
                break;
            if(S[j]['choices'][i] == 'electrical' and electrical.__len__() < C['electrical'] and
                S[j]['gpa'] >= 4.0 and S[j]['gpa'] < 13):
                electrical.append(S[j]['macid'])
                break;
            if(S[j]['choices'][i] == 'mechanical' and mechanical.__len__() < C['mechanical'] and
                S[j]['gpa'] >= 4.0 and S[j]['gpa'] < 13):
                mechanical.append(S[j]['macid'])
                break;
            if(S[j]['choices'][i] == 'software' and software.__len__() < C['software'] and
                S[j]['gpa'] >= 4.0 and S[j]['gpa'] < 13):
                software.append(S[j]['macid'])
                break;
            if(S[j]['choices'][i] == 'materials' and materials.__len__() < C['materials'] and
                S[j]['gpa'] >= 4.0 and S[j]['gpa'] < 13):
                materials.append(S[j]['macid'])
                break;
            if(S[j]['choices'][i] == 'engphys' and engphys.__len__() < C['engphys'] and
                S[j]['gpa'] >= 4.0 and S[j]['gpa'] < 13):
                engphys.append(S[j]['macid'])
                break;
            #If the student has not been allocated into anything they will be put into a
                department that is not empty temporarily
            if(i == 2):
                if(S[j]['gpa'] >= 4.0 and S[j]['gpa'] < 13):
                    print('WARNING! ', S[j]['macid'], ' has not been able to be allocated into any
                        their top 3 choices.', S[j]['macid'], ' will be placed in a temporary
                        department.')
                if(civil.__len__() < C['civil'] and S[j]['gpa'] >= 4.0 and S[j]['gpa'] < 13):
                    civil.append(S[j]['macid'])
                    break;
                if(chemical.__len__() < C['chemical'] and S[j]['gpa'] >= 4.0 and S[j]['gpa'] < 13):
                    chemical.append(S[j]['macid'])
                    break;
                if(electrical.__len__() < C['electrical'] and S[j]['gpa'] >= 4.0 and S[j]['gpa'] <
                    13):
```

```
                        electrical.append(S[j]['macid'])
                        break;
                    if(mechanical.__len__() < C['mechanical'] and S[j]['gpa'] >= 4.0 and S[j]['gpa'] <
                        13):
                        mechanical.append(S[j]['macid'])
                        break;
                    if(software.__len__() < C['software'] and S[j]['gpa'] >= 4.0 and S[j]['gpa'] < 13):
                        software.append(S[j]['macid'])
                        break;
                    if(materials.__len__() < C['materials'] and S[j]['gpa'] >= 4.0 and S[j]['gpa'] <
                        13):
                        materials.append(S[j]['macid'])
                        break;
                    if(engphys.__len__() < C['engphys'] and S[j]['gpa'] >= 4.0 and S[j]['gpa'] < 13):
                        engphys.append(S[j]['macid'])
                        break;

    stdntAllocations['civil'] = civil
    stdntAllocations['chemical'] = chemical
    stdntAllocations['electrical'] = electrical
    stdntAllocations['mechanical'] = mechanical
    stdntAllocations['software'] = software
    stdntAllocations['materials'] = materials
    stdntAllocations['engphys'] = engphys

    return stdntAllocations

freeChoice = readFreeChoice('freeChoice.txt')
deptCapacity = readDeptCapacity('deptCapacity.txt')
students = readStdnts('studentInfo.txt')

test = allocate(students, freeChoice, deptCapacity)
print(test)
```

H Code for testCalc.py

```
## @file testCalc.py
#  @Donisius Wigie
#  @A module to test CalcModule.py
#  @2019-01-12

from CalcModule import *
from ReadAllocationData import *
#A function to test if the sort function actually sorted the Students
def testSort(S):
    i = 0
    for i in range(S.__len__() - 1):
        if(S[i + 1]['gpa'] > S[i]['gpa']):
            print('The list is not sorted properly - FAILED')
            return 0
    print('The list is sorted properly - PASSED')
    return 1
#A function to obtain a list of macid's who have a gpa < 4.0, takes
#List of dictionaries produced by readStdnts
sample = readStdnts("studentInfo.txt")
sortedSample = sort(sample)
def failures(S):
    failureList = []
    for students in range(S.__len__()):
        if(S[students]['gpa'] < 4.0):
            failureList.append(S[students]['macid'])
    return failureList

freeChoice = readFreeChoice("freeChoice.txt")
dictList = readStdnts("studentInfo.txt")
deptCap = readDeptCapacity("deptCapacity.txt")
test = allocate(dictList, freeChoice, deptCap)
fails = failures(sample)

print('REGULAR INPUTS: ')
print('*********************************************************************************')
sortedSample = sort(sample)
test0 = testSort(sortedSample)
#A function to test whether or not the students who failed were allocated
#Takes dictionary of allocations and failure list
#Returns 1 if pass 0 if not
def belongOrNot(A, F):
    #allocate the students who belong
    for i in F:
        for j in range(A['civil'].__len__()):
            if(A['civil'][j] == i):
                print('belongOrNot test failed')
                return 0
        for j in range(A['chemical'].__len__()):
            if(A['chemical'][j] == i):
                print('belongOrNot test failed')
                return 0
        for j in range(A['electrical'].__len__()):
            if(A['electrical'][j] == i):
                print('belongOrNot test failed')
                return 0
        for j in range(A['mechanical'].__len__()):
            if(A['mechanical'][j] == i):
                print('belongOrNot test failed')
                return 0
        for j in range(A['software'].__len__()):
            if(A['software'][j] == i):
                print('belongOrNot test failed')
                return 0
        for j in range(A['materials'].__len__()):
            if(A['materials'][j] == i):
                print('belongOrNot test failed')
                return 0
        for j in range(A['engphys'].__len__()):
            if(A['engphys'][j] == i):
                print('belongOrNot test failed')
                return 0
        print('Students who have below 4.0 were not allocated - PASSED')
        return 1
print ('Students with below a 4.0 gpa: ', fails)
firstTest = belongOrNot(test, fails)
dictList = readStdnts("studentInfo.txt")
```

```
#A function to determine the first choice of the first choice students
def firstChoices(F, S):
    firstChoices = []
    for i in range(F.__len__()):
        for j in range(S.__len__()):
            if(F[i] == S[j]['macid'] and S[j]['gpa'] >= 4.0):
                firstChoices.append([F[i], S[j]['choices'][0]])
    return firstChoices

fcList = firstChoices(freeChoice, dictList)
try:
    test1 = allocate(dictList, freeChoice, deptCap)
    test1 = 1
    print('Regular inputs for allocate - PASSED')
except Exception as e:
    log.error("there was an exception %s", str(e))
    test1 = 0

print ('Inputs for average function: Regular inputs, ie. supplied files/binary genders')
try:
    test2 = average(dictList, 'male')
    test2 = average(dictList, 'female')
    test2 = 1
    print('Average function test - PASSED')
except Exception as e:
    log.error("there was an exception %s", str(e))
    test2 = 0

#A funtion to check if everyone who had free choice got
#first choice pick; takes dictionary of allocations and first choice list
def fcAllocated(A, fc):
    counter = fc.__len__()
    for i in range(fc.__len__()):
        for j in range(A['civil'].__len__()):
            if(fc[i][1] == 'civil' and fc[i][0] == A['civil'][j]):
                counter -= 1
        for j in range(A['chemical'].__len__()):
            if(fc[i][1] == 'chemical' and fc[i][0] == A['chemical'][j]):
                counter -= 1
        for j in range(A['electrical'].__len__()):
            if(fc[i][1] == 'electrical' and fc[i][0] == A['electrical'][j]):
                counter -= 1
        for j in range(A['mechanical'].__len__()):
            if(fc[i][1] == 'mechanical' and fc[i][0] == A['mechanical'][j]):
                counter -= 1
        for j in range(A['software'].__len__()):
            if(fc[i][1] == 'software' and fc[i][0] == A['software'][j]):
                counter -= 1
        for j in range(A['materials'].__len__()):
            if(fc[i][1] == 'materials' and fc[i][0] == A['materials'][j]):
                counter -= 1
        for j in range(A['engphys'].__len__()):
            if(fc[i][1] == 'engphys' and fc[i][0] == A['engphys'][j]):
                counter -= 1
    if(counter == 0):
        print('All free choices were granted to those who had free choice - PASSED')
        return 1
    print('fcAllocated test failed')
    return 0
print ('First choice students: ', fcList)
test3 = fcAllocated(test, fcList)

print('BOUNDARY CASES: ')
print('*******************************************************************************')
#Allocate function with empty department capacity file
deptCap = {}
print ('Input for Empty department test: ', deptCap, 'expected to not crash')
try:
    test4 = allocate(dictList, freeChoice, deptCap)
    test4 = 1
    print('Empty Department Test - PASSED')
except Exception as e:
    log.error("there was an exception %s", str(e))
    test4 = 0

#Allocate function with empty student list
dictList = []
print ('Input for Empty Student List Test: ', dictList, 'expected to not crash')
try:
```

```
    test5 = allocate(dictList, freeChoice, deptCap)
    test5 = 1
    print('Empty Student List Test - PASSED')
except Eception as e:
    log.error("there was an exception %s", str(e))
    test5 = 0

#Average function with a non-male/female gender Input
dictList = readStdnts("studentInfo.txt")
gender = 'apache attack helicopter'
print ('Input for gender - average function: ', gender, 'expected to not crash')
try:
    test6 = average(dictList, gender)
    test6 = 1
    print('Not A Real Gender Test - PASSED')
except Exception as e:
    log.error("there was an exception %s", str(e))
    test6 = 0
testInput = 'This is some random nonsense'

print ('input for allocation function: ', testInput, 'expected to not crash')
try:
    test7 = allocate(testInput, freeChoice, deptCap)
    test7 = 1
    print('Nonsense test 1 for allocate - PASSED')
except Exception as e:
    log.error("there was an exception %s", str(e))
    test7 = 0
#This is a test to see if the allocation function can handle students with negative gpa
print('Negative gpa test: ')
studentList = [{'macid': 'wigied', 'fname': 'Donisius', 'lname': 'Wigie', 'gender': 'male', 'gpa':
    -6.0, 'choices': ['software', 'electrical', 'engphys']}]
print('Input for studentList = ', studentList)
freeC = ['wigied']
try:
    test8 = allocate(studentList, freeC, deptCap)
    test8 = 1
    print('Negative gpa test for allocate - PASSED')
except Exception as e:
    log.error("there was an exception %s", str(e))
    test8 = 0
#This is a test to see if the allocation function can handle students with a gpa above 12
print('Excessively high gpa test: ')
studentList = [{'macid': 'wigied', 'fname': 'Donisius', 'lname': 'Wigie', 'gender': 'male', 'gpa':
    77.0, 'choices': ['software', 'electrical', 'engphys']}]
print('Input for studentList = ', studentList)
freeC = ['wigied']
try:
    test9 = allocate(studentList, freeC, deptCap)
    test9 = 1
    print('Excessively high gpa test for allocate - PASSED')
except Exception as e:
    log.error("there was an exception %s", str(e))
    test9 = 0
print('Empty list parameter for average function: ')
print('Input for average function: ', [])
try:
    test10 = average([], 'male')
    test10 = 1
    print('Empty list parameter for average function - PASSED')
except Exception as e:
    log.error("there was an exception %s", str(e))
    test10 = 0

total = test0 + test1 + test2 + test3 + test4 + test5 + test6 + test7 + test8 + test9 + test10
print('**********************************************************************************')
print('The total score is: ',total, 'out of', 11, 'tests passed: ', float((total/11) * 100), '%')
```

I Code for Partner's CalcModule.py

```
## @file CalcModule.py
#  @author Behzad Khamneli
#  @brief Sorts the students info based on their GPA, calculates their average and assigns them to a
     department.
#  @date 1/18/2019

## @brief Function sort takes a list of the dictionaries of students from readStdnts in
     ReadAllocationData.
#  @return A list of dictonaries in sorted order.
#  Citation: https://www.saltycrane.com/blog/2007/09/how-to-sort-python-dictionary-by-keys/
def sort(S):
        if S == "File Error":
                return "File Error"
        else:
                return sorted(S, key = lambda grade: grade['gpa'], reverse = True)

## @brief This function calcuates the average gpa of male or female students.
#  @details If param g is 'male', then it returns the average gpa of male students and if g is
     'female' then it returns the average gpa for female students. If non of them has been entered,
     then it returns an error.
#  @param L Is a list of dictionaries of students.
#  @param g Can be either male or female.
#  @return The average gpa of male or female students, with the choice depending on the param g.
def average(L, g):
        if L == "File Error":
                return "File Error"
        else:
                sumMale = 0
                numofMale = 0
                aveMale = 0

                sumFemale = 0
                numFemale = 0
                aveFemale = 0
                for dicty in L:
                        if dicty['gender'] == 'male':
                                sumMale = sumMale + dicty['gpa']
                                numofMale += 1
                        if dicty['gender'] == 'female':
                                sumFemale= sumFemale + dicty['gpa']
                                numFemale += 1
                if g == 'male':
                        aveMale = sumMale / numofMale
                        return aveMale
                if g == 'female':
                        aveFemale = sumFemale /numFemale
                        return aveFemale
                else:
                        return "g can be either male or female"
## @brief This function assigns the students to a department depending on their gpa and freechoice.
#  @details After each iteration, this function stores the macid of the students who have been
     assigned to a department to a list so that the same student will not be assigned to another
     department.
#  @param S Is a list of the dictionaries of students created by readStdnts.
#  @param F Is a list of students with free choice.
#  @param C Is a dictionary of department capacities.
#  @return A dictionary with the following format {'civil': [student, student,...], 'chemical':
     [student, student,...],...}. Student corresponds to the students' information.
def allocate(S, F, C):
        if (S == "File Error") or (F == "File Error") or (C == "File Error"):
                return "File Error"
        else:
                alldic = {}
                numofFree = len(F)
                assigned = []
                civil = []
                chemical = []
                electrical = []
                mechanical = []
                software = []
                materials = []
                engphys = []

                for std in sort(S):
                        for i in range (numofFree):
                                if std['macid'] == F[i]:
```

```
                        if std['choices'][0] == 'civil' and C['civil'] != 0:
                                civil.append(std)

                                C['civil'] = C['civil'] - 1

                        elif std['choices'][0] == 'chemical' and C['chemical'] != 0:
                                chemical.append(std)
                                C['chemical'] = C['chemical'] - 1

                        elif std['choices'][0] == 'electrical' and C['electrical'] !=
                             0:
                                electrical.append(std)
                                C['electrical'] = C['electrical'] - 1

                        elif std['choices'][0] == 'mechanical' and C['mechanical'] !=
                             0:
                                mechanical.append(std)
                                C['mechanical'] = C['mechanical'] - 1

                        elif std['choices'][0] == 'software' and C['software'] != 0:
                                software.append(std)
                                C['software'] = C['software'] - 1

                        elif std['choices'][0] == 'materials' and C['materials'] != 0:
                                materials.append(std)
                                C['materials'] = C['materials'] - 1

                        elif std['choices'][0] == 'engphys' and C['engphys'] != 0:
                                engphys.append(std)
                                C['engphys'] = C['engphys'] - 1

for std in sort(S):
        for i in range(3):
                if std['macid'] not in F and std['macid'] not in assigned and
                     std['gpa'] >= 4.0:

                        if std['choices'][i] == 'civil':
                                if C['civil'] != 0:
                                        civil.append(std)
                                        C['civil'] = C['civil'] - 1
                                        assigned.append(std['macid'])


                        if std['choices'][i] == 'chemical':
                                if C['chemical'] != 0:
                                        chemical.append(std)
                                        C['chemical'] = C['chemical'] - 1
                                        assigned.append(std['macid'])

                        if std['choices'][i] == 'electrical':
                                if C['electrical'] != 0:
                                        electrical.append(std)
                                        C['electrical'] = C['electrical'] - 1
                                        assigned.append(std['macid'])

                        if std['choices'][i] == 'mechanical':
                                if C['mechanical'] != 0:
                                        mechanical.append(std)
                                        C['mechanical'] = C['mechanical'] - 1
                                        assigned.append(std['macid'])

                        if std['choices'][i] == 'software':
                                if C['software'] != 0:
                                        software.append(std)
                                        C['software'] = C['software'] - 1
                                        assigned.append(std['macid'])

                        if std['choices'][i] == 'materials':
                                if C['materials'] != 0:
                                        materials.append(std)
                                        C['materials'] = C['materials'] - 1
                                        assigned.append(std['macid'])

                        if std['choices'][i] == 'engphys':
                                if C['engphys'] != 0:
                                        engphys.append(std)
                                        C['engphys'] = C['engphys'] - 1
                                        assigned.append(std['macid'])

alldic['civil'] = civil
```

```
alldic['chemical'] = chemical
alldic['electrical'] = electrical
alldic['mechanical'] = mechanical
alldic['software'] = software
alldic['materials'] = materials
alldic['engphys'] = engphys
return alldic
```

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
```