Assignment 1 Solution

Michael Barreiros barreirm

January 25, 2019

The purpose of the program created in Assignment 1 was to read and process data from first-year engineering students and to allocate them into one of their three choices if possible.

1 Testing of the Original Program

The rationale for my test cases for the module CalcModule was that all problems that might have occurred during reading data were dealt with in the ReadAllocationData module.

Some problems could have been that:

- GPAs are out of the 0.0-12.0 range
- Choices were not of the choices available
- Any issues with capitalization
- Any mismatch of data types or invalid input

With this in mind my CalcModule passed all my test cases. My testCalc Module had 10 test cases.

1.1 Testing for Average

This includes 4 tests. Testing the average of a student list for each gender. Then testing the average of a student list where no member of the gender exists, again this was done for each gender. An assumption was made when testing for the average of a gender with no members that the average function would return 0 and not None.

1.2 Testing for Sort

This includes 4 tests. Testing for the sorting of students all with unique GPAs. Testing for students with some of the students sharing the same GPA. It is assumed that students with the same GPA get sorted in the way they appear in the original student list. Next, I tested the sorting on an empty list, the expected result was an empty list. The last test for Sort was testing an already sorted list. The expected result is to have the same list returned.

1.3 Testing for Allocate

This includes 2 tests. The first test really was testing many things. The first test of allocation tested for the allocation of a free choice student that has a GPA lower than someone without free choice, it is expected that the free choice student gets allocated first. The next test here tested the allocation of a student with a 4.0 GPA, I made the assumption that any student with a GPA lower than or equal to a 4.0 does not get allocated. The next test here tested to see what would happen when students apply for a department with not enough space to accommodate everyone. I made the assumption that even if a department goes over capacity with all free choice students the department cannot allocate more students than their initial capacity. Therefore in this test, we expect any student applying for a department who has reached their capacity to move on to their next choice. The final test here was to test what happens when a free choice student has a GPA below 4.0. As stated earlier before we expect all students with a GPA at or below a 4.0 to not get allocated, this includes students with free choice. The second test was a test for the allocation of zero students, the expected result was a dictionary with departments as keys and an empty list as each of the key's value.

2 Results of Testing Partner's Code

2.1 First Attempt

I was unable to test my partner's code because he implemented two modules named al_constants.py and al_utility.py. My partner used al_contants to implement MIN_GPA, MAX_GPA, MIN_PASSING_GPA, GENDERS functions. My partner used al_utility to implement the remove_duplicates.

2.2 Second Attempt

Steps taken to get the program to run:

1. Commented out the two lines of import statements
2. Made variables for MIN_GPA, MAX_GPA, MIN_PASSING_GPA with values 0.0, 12.0, and 4.0 respectively
3. Commented out remove_duplicates(students) from the function allocate(S, F, C)

The program ran, three of my tests failed.
The tests that failed include:

1. Testing for male average when there exists no males
2. Testing for female average when there exists no females
3. Testing allcation of students. The particular part that failed was the allocation of a student with a 4.0 GPA

3 Discussion of Test Results

3.1 Problems with Original Code

Through my test cases, my code passed every one of them. I cannot say that there are zero problems based on my code because a lot of the reasoning behind the decisions made on how to code something or what to do in a certain scenario came from assumptions that I had to make.

I would say that the majority of problems arise from ReadAllocationData and the assumptions I had to make when creating the needed functions.

For readStdnts, the assumption was made that in the file every student appeared in a new line and had entries for "macid fname lname gender GPA choice1 choice2 choice3" only appearing in that exact order with spaces between each entry and did not include the key's needed for their dictionary counterparts. This meant that I did not expect "madId: barreirm" I expected only "barreirm". When creating the function I also accounted for some special characters like "{", "}", """, ",", "[", and "]".

For readFreeChoice, the assumption that was made was that only students with free choice appear in the file and that the student's macId is in the file. They could all be on one line but must be separated by at least one space, they could also all be on separate lines.

For readDeptCapacity the assumption that was made was that in the input file the department's name comes first then the capacity for that department separated by at least one space. Data entries could all be on separate lines or on one line or a mixture of the two options. This function accounted for some special characters like ":" and tabbing.

If I were to return to this function I would have accounted for "{" and "}" just in case the incoming file tried to structure their data like a dictionary already.

3.2 Problems with Partner's Code

My partner's CalcModule failed three of my tests. The reason for failing the first two items on the list of failed tests was that my partner returned None rather than 0 when there were no members of the gender that you were trying to find the average for. This wasn't a problem with the code just a difference in assumptions.

The third item on the list of failed tests occurred because my partner assumed that students that have a 4.0 GPA still get allocated while I made the assumption that they do not. Again, this was not a problem with their code, just a difference of assumptions. I made the assumption based on the design specifications for the function allocate(S,F,C) where it states "The algorithm for the allocation will allocate all students with a GPA greater than 4.0." I interpreted that as meaning anyone that doesn't have a GPA greater than 4.0, this includes 4.0, will not get allocated.

4 Critique of Design Specification

One general critique of the design specifications is that it was very ambiguous and it required the designer to make a lot of assumptions. If this was a one person project then it might not be a big deal but if it were a project where many people were working on it and their individual modules or programs had to work together then all ambiguities must be dealt with in the design specification. Through testing my own program and testing someone else's program I can see that we made different assumptions in some cases which resulted in different outputs in some scenarios.

I think that the for ReadAllocationData the design specification should have told us exactly what to expect as input when it comes to the structure of the file and the data within each file. This would allow us to deal with the incoming data appropriately. This would make the process of creating this method less tedious because we wouldn't have to think about all possible scenarios of incoming data and how we would deal with it if ever encountered.

For CalcModule the design specification should have made it more clear what to do when encountering certain scenarios such as what to do with a student who has a GPA of exactly 4.0. Another thing that the design specification should have told us is what happens when free choice students all apply to a certain department and go over its capacity. It should tell us whether or not the department would allow for its capacity to increase since a promise was made to free choice students that they will get their

first choice. Another thing that should be specified is what to return when computing the average when there is no population for that gender, is it 0? Is it None? Another ambiguity is what happens to students where all three choices are filled. Do they get allocated to a random department that has space? If so how does the randomization of departments occur? Do they get a list of open departments to make a new list of choices and then start the allocation process again with other students who did not get allocated?

5 Answers to Questions

(a) I would make average(L,g) more general by allowing for the computation of average to occur over different filters. Meaning I would allow for the computation of average for a certain department or based on student's first choice. I would also allow to compute the average of students who's first name begin with a certain letter. The same could be said for their macid's and their last name. As for sort I would allow for the option of sorting in increasing or decreasing order. I would also include the ability to sort by more than just by GPA. I would allow for the ability to sort be alphabetical order for all options of macid, first name, or last name. I would also allow for the ability to sort by the first choice. With all of this I would also allow for the combination of different kinds of sorting. Such as sorting first by the department that is their first choice then sort by GPA.

(b) Aliasing in this context means having different variables referring to the same dictionary. The dictionary referred to by the two variables is said to have two aliases. Since dictionaries are mutable data structures this can be a concern. If you try to copy a dictionary by simply creating a new dictionary this may cause problems. If you change the values of keys or add/delete keys in this new "copy" it will affect the original dictionary that you tried to copy as well. In order to guard against this one would use deepcopy in order to create a completely independent copy of the dictionary.

(c) A potential test case would have been reading students for readStdnts who's data weren't in the right location, such as their GPA being where fname should be or if some entries were empty, such as not having a third choice. A potential test case for all three functions would have been to test for garbage input. This could include random characters or incorrect department names. A potential test case for all three functions could have been to read from an empty file. Another test could have been to try to read from a file that does not exist. I believe CalcModule.py was selected over ReadAllocationData.py as the one we should test because there were fewer variations and assumptions that would have to be made when creating the CalcModule module. Also, a lot of the "user" input errors that would have to be considered when testing

ReadAllocationData don't exist when testing for CalcModule since the assumption is made that all "user" input errors were taken care of in ReadAllocationData.

(d) The problems with using strings in this way is that the key or the value would have to be inputted exactly as it is shown in the dictionary and would allow for zero variations such as a capital letter in the wrong position. A better approach would be to use named tuples like Professor Smith suggested in class. This allows us to create a set of key values to be used for the keys of our dictionary and in the 'male', 'female' example this allows us to create a set of expected values for gender and we can access that set by having the fieldname of the set be gender.

(e) The mathematic definition of a tuple is a collection of elements of possibly different types. Each tuple has one or more fields and each field has a unique identifier called the field name. By this definition, dictionaries, named tuple, and a custom class all can be used to implement the mathematical notion of tuples in python. I would recommend changing students to namedtuple instead of a dictionary. Each student's choices can be implemented using a namedtuple. I think that changing to namedtuples is a good idea because the students and their choices will not change after creation so that the fact that tuples are immutable.

(f) If the list of strings was changed to a different data structure, like a tuple, then CalcModule wouldn't have to be modified as I iterated over the entries of the list. You iterate over the entries of a tuple in the same way. I have never worked with tuples but that is my understanding. I do not believe that a custom class would have to be modified if its data structure changed. The way I coded my scan through the students' choices was by running a for loop that iterated over the first three indexes of the choice's list. If a custom class were to return true when trying to access a choice that is not there then it would not change anything since I would never be reaching that point.

F Code for ReadAllocationData.py

```
## @file ReadAllocationData.py
#  @author Michael Barreiros
#  @title ReadAllocationData
#  @date 01/17/19

## @brief This function takes an input file of students and outputs a list of dictionaries of students
#  @details This function takes an input file such as students.txt and reads the data from it. It will
#  strip any special characters from it and create the list in the proper format. An assumption was
#  made that every student is on a new line and that the information is given in only one order "macid
#  fname lname gender gpa choice1 choice2 choice3"
#  @param s string of a file name to be opened and read
#  @return Students [{student},{student},...,{student}] list of dictionaries. Each dictionary is the
    information for one student
def readStdnts(s):
        f = open(s, "r")

        StudentData = f.read()
        StudentData = StudentData.replace("{", "")
        StudentData = StudentData.replace("}", "")
        StudentData = StudentData.replace("\"", "")
        StudentData = StudentData.replace("[", "")
        StudentData = StudentData.replace("]", "")
        StudentData = StudentData.replace(",", "")
        Students = StudentData.splitlines()

        ## This for loop creates a new dictionary and populates it for the current student and places
             it in the list
        for i in range(0,len(Students)):
                Students[i] = Students[i].lower()
                Students[i] = Students[i].split()

                choices = [Students[i][5] , Students[i][6], Students[i][7]]

                StdntDict = {'macid': Students[i][0], 'fname': Students[i][1], 'lname':
                     Students[i][2], 'gender': Students[i][3], 'gpa': Students[i][4], 'choices':
                     choices }

                StdntDict['gpa'] = float(StdntDict['gpa'])

                Students[i] = StdntDict

        f.close()
        return Students

## @brief This function reads from an input file and outputs a list of macId's
#  @details This function reads from an input file. An assumption is made that in this file only the
    macId of students
#  who received free choice appear, it does not matter if they are on newlines, or if some special
    characters appear.
#  After reading and stripping unneeded characters the data is formatted into a list who's entries are
    macId's
#  @param s string of a file name to be opened and read
#  @return names a list who's entries are macId's of student's who got free choice
def readFreeChoice(s):
        f = open(s, "r")

        names = f.read()
        names = names.strip()
        names = names.replace(",", "")
        names = names.replace(" ", "\n")
        names = names.lower()
        names = names.splitlines()

        ## emptyElements was added to control the input and get rid of any empty elements that might
             have occured
        emptyElements = []
        for i in range(0, len(names)):
                if names[i] == '':
                        emptyElements.append(i)

        for j in range(0, len(emptyElements)):
                del names[emptyElements[j]-j]

        f.close()
        return names
```

```
## @brief This function reads from an input file and outputs a dictionary with departments as keys and
    their capacity as values
#  @details This function reads from an input file. An assumption is made that in the input file the
    department name comes first
#  then the department's capacity comes next with at least one space between. Some expected special
    characters get stripped.
#  The function then matches the department name and updates the capacity value in an existing
    dictionary.
#  @param s string of a file name to be opened and read
#  @return Capacity {'department': capacity} A dictionary of departments as keys and their capacity as
    values
def readDeptCapacity(s):
        f = open(s, "r")

        capacities = f.read()
        capacities = capacities.strip()
        capacities = capacities.replace(":", "")
        capacities = capacities.replace("\t", "")
        capacities = capacities.replace(",", "")
        capacities = capacities.replace(" ", "\n")
        capacities = capacities.lower()
        capacities = capacities.splitlines()

        ## emptyElements was added to control the input and get rid of any empty elements that might
            have occured
        emptyElements = []
        for i in range(0, len(capacities)):
                if capacities[i] == '':
                        emptyElements.append(i)

        for j in range(0, len(emptyElements)):
                del capacities[emptyElements[j]-j]

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

        for i in range(0, len(capacities), 2):
                if capacities[i] == 'civil':
                        Capacity['civil'] = int(capacities[i+1])
                if capacities[i] == 'chemical':
                        Capacity['chemical'] = int(capacities[i+1])
                if capacities[i] == 'electrical':
                        Capacity['electrical'] = int(capacities[i+1])
                if capacities[i] == 'mechanical':
                        Capacity['mechanical'] = int(capacities[i+1])
                if capacities[i] == 'software':
                        Capacity['software'] = int(capacities[i+1])
                if capacities[i] == 'materials':
                        Capacity['materials'] = int(capacities[i+1])
                if capacities[i] == 'engphys':
                        Capacity['engphys'] = int(capacities[i+1])

        return Capacity
```

G Code for CalcModule.py

```
## @file CalcModule.py
#  @author Michael Barreiros
#  @title CalcModule
#  @date 01/17/19

## @brief This function sorts a list of dictionaries by the key gpa in decending order
#  @param S A list of students. Each student is a dictionary.
#  @return newList The same list that was inputted but now sorted in decending order by gpa
def sort(S):
        ## newList was sorted using a line of code that was found on stackoverflow
        ## link is
             https://stackoverflow.com/questions/72899/how-do-i-sort-a-list-of-dictionaries-by-a-value-of-the-dictionary
        newList = sorted(S, key = lambda k: k['gpa'], reverse = True)
        return newList

## @brief This function returns the average gpa of a gender in the inputted list. Returns 0 if no
     population for that gender.
#  @param L A list of students. Each student is a dictionary.
#  @param g A string which would either be "male" or "female"
#  @return average The average gpa for the selected gender
def average(L, g):
        average = 0.0
        sumOfGpa = 0.0
        numOfStudents = 0.0

        for student in L:
                if student['gender'] == g.lower():
                        sumOfGpa += student['gpa']
                        numOfStudents += 1

        if numOfStudents == 0:
                return 0
        else:
                average = sumOfGpa/numOfStudents
                return average

## @brief This is a function allocates students to the appropriate department
#  @details The function sorts the incoming student list by gpa. Then creates new lists for each of
     the departments.
#  These lists are populated with new students, first from students with free choice then with
     students from
#  the original list of students. The function returns a dictionary with departments as keys and a
     list of students,
#  each student is represented by their own dictionary.
#  An assumption is made that students with free choice who's choice has reached its capacity would
     attempt to get
#  allocated to their next choice.
#  @param S A list of students. Each student is a dictionary.
#  @param F A list of student's macID's. Each element of the list is a string
#  @param C A Dictionary with departments as key values and thier capacities as values
#  @return allocatedDictionary {'department': [student, student,...], ...} A dictionary with
     departments as keys and a list of students as values
def allocate(S, F, C):
        #Allocation algorithm assumes that students with free choice are still allocated based on gpa
        S = sort(S)

        civilList = []
        chemList = []
        elecList = []
        mechList = []
        softList = []
        matList = []
        engphysList = []

        # this will be a list of only free choice students so they can be allocated first
        freeChoiceStudents = []

        for student in S:
                if student['macid'] in F: #student has free choice
                        freeChoiceStudents.append(student)

        for student in freeChoiceStudents:
                if student['gpa'] > 4.0: #only allocate student if gpa is greater than 4.0
                        for i in range(0,3): #runs through each of their choices only if their
                             previous choice was full
```

```
                        # each of these if statements try to match student choice to one of
                            the departments
                        # then it checks to see if the department is full. If full move on to
                            next choice
                        if student['choices'][i] == 'civil':
                                if len(civilList) < C['civil']:
                                        civilList.append(student)
                                        break
                                else:
                                        continue
                        if student['choices'][i] == 'chemical':
                                if len(chemList) < C['chemical']:
                                        chemList.append(student)
                                        break
                                else:
                                        continue
                        if student['choices'][i] == 'electrical':
                                if len(elecList) < C['electrical']:
                                        elecList.append(student)
                                        break
                                else:
                                        continue
                        if student['choices'][i] == 'mechanical':
                                if len(mechList) < C['mechanical']:
                                        mechList.append(student)
                                        break
                                else:
                                        continue
                        if student['choices'][i] == 'software':
                                if len(softList) < C['software']:
                                        softList.append(student)
                                        break
                                else:
                                        continue
                        if student['choices'][i] == 'materials':
                                if len(matList) < C['materials']:
                                        matList.append(student)
                                        break
                                else:
                                        continue
                        if student['choices'][i] == 'engphys':
                                if len(engphysList) < C['engphys']:
                                        engphysList.append(student)
                                        break
                                else:
                                        continue
        else:
                continue


for student in S:
        if student['gpa'] > 4.0: # only allocates student if gpa is greater than 4/0
                if student['macid'] in F: # this is here to check for already allocated
                    members of this list
                        continue

                for i in range(0,3):
                        if student['choices'][i] == 'civil':
                                if len(civilList) < C['civil']:
                                        civilList.append(student)
                                        break
                                else:
                                        continue
                        if student['choices'][i] == 'chemical':
                                if len(chemList) < C['chemical']:
                                        chemList.append(student)
                                        break
                                else:
                                        continue
                        if student['choices'][i] == 'electrical':
                                if len(elecList) < C['electrical']:
                                        elecList.append(student)
                                        break
                                else:
                                        continue
                        if student['choices'][i] == 'mechanical':
                                if len(mechList) < C['mechanical']:
                                        mechList.append(student)
                                        break
                                else:
```

```
                                        continue
                                if student['choices'][i] == 'software':
                                        if len(softList) < C['software']:
                                                softList.append(student)
                                                break
                                        else:
                                                continue
                                if student['choices'][i] == 'materials':
                                        if len(matList) < C['materials']:
                                                matList.append(student)
                                                break
                                        else:
                                                continue
                        if student['choices'][i] == 'engphys':
                                        if len(engphysList) < C['engphys']:
                                                engphysList.append(student)
                                                break
                                        else:
                                                continue
                else:
                        continue

        allocatedDictionary = {'civil': civilList, 'chemical': chemList, 'electrical': elecList,
            'mechanical': mechList, 'software': softList, 'materials': matList, 'engphys':
            engphysList}

        return allocatedDictionary
```

H Code for testCalc.py

```
## @file testCalc.py
#  @author Michael Barreiros
#  @brief
#  @date 01/16/2019

from ReadAllocationData import *
from CalcModule import *

def compare(test, result, name):
        if test == result:
                print("Testing: ", name)
                print("Test passed: %s == %s" %(test, result))
        else:
                print("Testing: ", name)
                print("Test failed: %s != %s" %(test, result))

        print()

def testAverageMales():
        students = [{'macid': 'barreirm', 'fname': 'michael', 'lname': 'barreiros', 'gender': 'male',
             'gpa': 9.0, 'choices': ['software', 'mechanical', 'electrical']},
        {'macid': 'teststdnt1', 'fname': 'joe', 'lname': 'shmoe', 'gender': 'male', 'gpa': 6.0,
             'choices': ['materials', 'engphys', 'civil']},
        {'macid': 'teststdnt2', 'fname': 'kate', 'lname': 'makabali', 'gender': 'female', 'gpa': 12.0,
             'choices': ['chemical', 'mechanical', 'civil']},
        {'macid': 'teststdnt3', 'fname': 'bella', 'lname': 'alleb', 'gender': 'female', 'gpa': 4.7,
             'choices': ['electrical', 'software', 'engphys']}]
        test = average(students, 'male')

        compare(test, 7.5, "Male average")

def testAverageFemales():
        students = [{'macid': 'barreirm', 'fname': 'michael', 'lname': 'barreiros', 'gender': 'male',
             'gpa': 9.0, 'choices': ['software', 'mechanical', 'electrical']},
        {'macid': 'teststdnt1', 'fname': 'joe', 'lname': 'shmoe', 'gender': 'male', 'gpa': 6.0,
             'choices': ['materials', 'engphys', 'civil']},
        {'macid': 'teststdnt2', 'fname': 'kate', 'lname': 'makabali', 'gender': 'female', 'gpa': 12.0,
             'choices': ['chemical', 'mechanical', 'civil']},
        {'macid': 'teststdnt3', 'fname': 'bella', 'lname': 'alleb', 'gender': 'female', 'gpa': 4.7,
             'choices': ['electrical', 'software', 'engphys']}]
        test = average(students, 'female')

        compare(test, 8.35, "Female average")

def testAverageNoMales():
        students = [{'macid': 'teststdnt2', 'fname': 'kate', 'lname': 'makabali', 'gender': 'female',
             'gpa': 12.0, 'choices': ['chemical', 'mechanical', 'civil']},
        {'macid': 'teststdnt3', 'fname': 'bella', 'lname': 'alleb', 'gender': 'female', 'gpa': 4.7,
             'choices': ['electrical', 'software', 'engphys']}]
        test = average(students, 'male')

        compare(test, 0, "Male average of a student list with no males")

def testAverageNoFemales():
        students = [{'macid': 'barreirm', 'fname': 'michael', 'lname': 'barreiros', 'gender': 'male',
             'gpa': 9.0, 'choices': ['software', 'mechanical', 'electrical']},
        {'macid': 'teststdnt1', 'fname': 'joe', 'lname': 'shmoe', 'gender': 'male', 'gpa': 6.0,
             'choices': ['materials', 'engphys', 'civil']}]
        test = average(students, 'female')

        compare(test, 0, "Female average of a student list with no females")

def testSortNormal():
        students = [{'macid': 'barreirm', 'fname': 'michael', 'lname': 'barreiros', 'gender': 'male',
             'gpa': 9.0, 'choices': ['software', 'mechanical', 'electrical']},
        {'macid': 'teststdnt1', 'fname': 'joe', 'lname': 'shmoe', 'gender': 'male', 'gpa': 6.0,
             'choices': ['materials', 'engphys', 'civil']},
        {'macid': 'teststdnt2', 'fname': 'kate', 'lname': 'makabali', 'gender': 'female', 'gpa': 12.0,
             'choices': ['chemical', 'mechanical', 'civil']},
        {'macid': 'teststdnt3', 'fname': 'bella', 'lname': 'alleb', 'gender': 'female', 'gpa': 4.7,
             'choices': ['electrical', 'software', 'engphys']}]

        sortedStudents = [{'macid': 'teststdnt2', 'fname': 'kate', 'lname': 'makabali', 'gender':
             'female', 'gpa': 12.0, 'choices': ['chemical', 'mechanical', 'civil']},
        {'macid': 'barreirm', 'fname': 'michael', 'lname': 'barreiros', 'gender': 'male', 'gpa': 9.0,
             'choices': ['software', 'mechanical', 'electrical']},
```

```
        {'macid': 'teststdnt1', 'fname': 'joe', 'lname': 'shmoe', 'gender': 'male', 'gpa': 6.0,
            'choices': ['materials', 'engphys', 'civil']},
        {'macid': 'teststdnt3', 'fname': 'bella', 'lname': 'alleb', 'gender': 'female', 'gpa': 4.7,
            'choices': ['electrical', 'software', 'engphys']}]

        test = sort(students)

        compare(test, sortedStudents, "Sorting list of students with unique GPAs")

def testSortSame():
        students = [{'macid': 'barreirm', 'fname': 'michael', 'lname': 'barreiros', 'gender': 'male',
            'gpa': 9.0, 'choices': ['software', 'mechanical', 'electrical']},
        {'macid': 'teststdnt1', 'fname': 'joe', 'lname': 'shmoe', 'gender': 'male', 'gpa': 4.7,
            'choices': ['materials', 'engphys', 'civil']},
        {'macid': 'teststdnt2', 'fname': 'kate', 'lname': 'makabali', 'gender': 'female', 'gpa': 12.0,
            'choices': ['chemical', 'mechanical', 'civil']},
        {'macid': 'teststdnt3', 'fname': 'bella', 'lname': 'alleb', 'gender': 'female', 'gpa': 4.7,
            'choices': ['electrical', 'software', 'engphys']}]

        sortedStudents = [{'macid': 'teststdnt2', 'fname': 'kate', 'lname': 'makabali', 'gender':
            'female', 'gpa': 12.0, 'choices': ['chemical', 'mechanical', 'civil']},
        {'macid': 'barreirm', 'fname': 'michael', 'lname': 'barreiros', 'gender': 'male', 'gpa': 9.0,
            'choices': ['software', 'mechanical', 'electrical']},
        {'macid': 'teststdnt1', 'fname': 'joe', 'lname': 'shmoe', 'gender': 'male', 'gpa': 4.7,
            'choices': ['materials', 'engphys', 'civil']},
        {'macid': 'teststdnt3', 'fname': 'bella', 'lname': 'alleb', 'gender': 'female', 'gpa': 4.7,
            'choices': ['electrical', 'software', 'engphys']}]

        test = sort(students)

        compare(test, sortedStudents, "Sorting list of students some with same GPAs")

def testSortEmpty():
        students = []

        sortedStudents = []

        test = sort(students)

        compare(test, sortedStudents, "Sorting an empty list of students")

def testSortSorted():
        students = [{'macid': 'barreirm', 'fname': 'michael', 'lname': 'barreiros', 'gender': 'male',
            'gpa': 12.0, 'choices': ['software', 'mechanical', 'electrical']},
        {'macid': 'teststdnt1', 'fname': 'joe', 'lname': 'shmoe', 'gender': 'male', 'gpa': 11.0,
            'choices': ['materials', 'engphys', 'civil']},
        {'macid': 'teststdnt2', 'fname': 'kate', 'lname': 'makabali', 'gender': 'female', 'gpa': 10.0,
            'choices': ['chemical', 'mechanical', 'civil']},
        {'macid': 'teststdnt3', 'fname': 'bella', 'lname': 'alleb', 'gender': 'female', 'gpa': 9.0,
            'choices': ['electrical', 'software', 'engphys']}]

        sortedStudents = [{'macid': 'barreirm', 'fname': 'michael', 'lname': 'barreiros', 'gender':
            'male', 'gpa': 12.0, 'choices': ['software', 'mechanical', 'electrical']},
        {'macid': 'teststdnt1', 'fname': 'joe', 'lname': 'shmoe', 'gender': 'male', 'gpa': 11.0,
            'choices': ['materials', 'engphys', 'civil']},
        {'macid': 'teststdnt2', 'fname': 'kate', 'lname': 'makabali', 'gender': 'female', 'gpa': 10.0,
            'choices': ['chemical', 'mechanical', 'civil']},
        {'macid': 'teststdnt3', 'fname': 'bella', 'lname': 'alleb', 'gender': 'female', 'gpa': 9.0,
            'choices': ['electrical', 'software', 'engphys']}]

        test = sort(students)

        compare(test, sortedStudents, "Sorting an already sorted list")

def testAllocateNormal():
        # in this function we have several scenarios being tested.
        # what happens when someone with free choice has a lower gpa than someone with free choice.
            free choice gets allocated first
        # we also have a student with a 4.0 gpa. We expect them not to get allocated
        # we have a department with not enough space to allocate everyone so some students have to be
            allocated to their next choices
        # we have a student with a gpa lower than a 4.0 and has free choice. We expect them not to get
            allocated
        students = [{'macid': 'barreirm', 'fname': 'michael', 'lname': 'barreiros', 'gender': 'male',
            'gpa': 12.0, 'choices': ['software', 'mechanical', 'electrical']},
        {'macid': 'freeChoiceLowGpa', 'fname': 'joe', 'lname': 'shmoe', 'gender': 'male', 'gpa': 2.0,
            'choices': ['materials', 'engphys', 'civil']},
        {'macid': '4.0', 'fname': 'kate', 'lname': 'makabali', 'gender': 'female', 'gpa': 4.0,
            'choices': ['chemical', 'mechanical', 'civil']},
```

```
        {'macid': 'freeChoice', 'fname': 'bella', 'lname': 'alleb', 'gender': 'female', 'gpa': 9.0,
            'choices': ['software', 'electrical', 'engphys']}]

        freeChoice = ["freeChoice", "freeChoiceLowGpa"]

        deptCap = {'civil': 1, 'chemical': 1, 'electrical': 1, 'mechanical': 1, 'software': 1,
            'materials': 1, 'engphys': 1}

        test = allocate(students, freeChoice, deptCap)

        allocatedStudents = {'civil': [], 'chemical': [], 'electrical': [],
        'mechanical': [{'macid': 'barreirm', 'fname': 'michael', 'lname': 'barreiros', 'gender':
            'male', 'gpa': 12.0, 'choices': ['software', 'mechanical', 'electrical']}],
        'software': [{'macid': 'freeChoice', 'fname': 'bella', 'lname': 'alleb', 'gender': 'female',
            'gpa': 9.0, 'choices': ['software', 'electrical', 'engphys']}],
        'materials': [], 'engphys': []}

        compare(test, allocatedStudents,
        "Allocation of student list containing \n" +
        "1) Free Choice Student taking the spot of non-free-choice\n" +
        "2) Student below a 4.0 with free choice not getting allocated\n" +
        "3) A student with exactly a 4.0 not getting allocated")

def testAllocationEmpty():
        students = []
        freeChoice = []
        deptCap = {'civil': 1, 'chemical': 1, 'electrical': 1, 'mechanical': 1, 'software': 1,
            'materials': 1, 'engphys': 1}

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

        test = allocate(students,freeChoice, deptCap)

        compare(test, allocatedStudents, "Allocation of an empty student list")


def test():
        testAverageMales()
        testAverageFemales()
        testAverageNoMales()
        testAverageNoFemales()
        testSortNormal()
        testSortSame()
        testSortEmpty()
        testSortSorted()
        testAllocateNormal()
        testAllocationEmpty()

test()
```

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

''' Commenting out import statement to get partner files to work - by Michael Barreiros
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):
        MIN_PASSING_GPA = 4.0 #CREATED VARIABLES TO GET PROGRAM TO RUN -Michael Barreiros
        MAX_GPA = 12.0

    if not (MIN_PASSING_GPA <= student['gpa'] <= MAX_GPA):
        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):
        MIN_GPA = 0.0 #CREATED VARIABLES TO GET PROGRAM TO RUN - Michael Barreiros
        MAX_GPA = 12.0
    if not L:
        return None
    gpas = []
    for student in L:
        if not (MIN_GPA <= student['gpa'] <= MAX_GPA):
            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
```