Assignment 1 Report

Zayed Sheet, Sheetz

January 29, 2019

The following document is a thorough report on the 2AA4 Assignment 1. The purpose of this software design exercise is to write a python program that uses three modules named ReadAllocationData.py, CalcModule.py and testCalc.py to take an input of students in first year engineering and place them in one of their desired programs of choice depending on whether they meet the requirements for that program.

1 Testing of the Original Program

When creating my test cases, the first thing I did was try to cover all the basic cases that essentially assess the correctness of the code. These basic cases would have expected inputs and expected outputs to test the basic functionality of the code. I then added some cases that test the robustness of the program by essentially trying to break the code with unexpected inputs. For example, cases where the inputs would contain empty lists, empty dictionaries or unrecognized string literals. My test cases go as follows:

While developing my test cases, the only issue I ran into was an unexpected output due to one of my global variables being changed in another test case. Other than that every test case passed, presumably because I was testing my code as I was writing it, and because my insight didn't really change much while writing my test cases, since writing my code. My test cases go as follows:

1. sortList

   The purpose of this test case is to test the basic functionality of the sort function. A list of dictionaries is passed into the function and the expected output is a list of dictionaries sorted by the 'gpa' key in descending order.

2. emptySortAllocate

This test case tests the robustness of the program when empty parameters are passed in. There are two parts to this test case. First it checks if the sort function works when an empty list is passed in. The expected output for this is an empty list. If this works then it checks if the allocate function works when the parameters are empty. Because the allocate function takes several parameters, the test case checks three different combinations of parameters to see if it still works when one parameter is empty but others are not. The reason I combined two test cases in this function is because if the sort case fails, then the allocate function will fail anyways.

3. emptyGender

   This test case tests what happens if an empty list is passed into the average function. The rationale being that the user may have entered the wrong list as an input but the program should still run without error.

4. wrongGender

   This test case tests how the program handles unexpected inputs. In this test case the gender parameter into the average function is ’test’ rather than ’male’ or ’female’. The rationale behind this is in-case the programer or user mistypes male or female or enters any other input for the gender parameter. The expected output is -1.

5. zeroGender

   The purpose for this test case is to see what happens when the list of students doesn’t have a single ’male’ and/or ’female’. For example, if all the students were male but female is passed into the function to test for the average gpa. The rationale behind this test case is to ensure there is no zero by division error despite there being a reasonable input. The expected output is -1.

6. allPassAllocate

   This is one of many test cases to test the basic functionality of the allocate function. I created several test cases for allocate to pinpoint where the problem is if an error did occur. This as well as the rest of my test cases are to test for correctness because the allocate function has several aspects to it. In this test case all students pass and are allocated to their first choice. There are no capacities or free choice students.

7. failAllocate

   In this test case some students fail, and therefore should not be present in the list outputted by the function. This test case tests if the program still works when students fail. There are no free choice students, no capacity and everyone gets their first choice if they passed.

8. freeChoiceAllocate

   In this test there are free choice students, all students are allocated, capacities are limited however all students pass. Free choice students should always get their first choice and should be entered into their program before other students. Subsequently, some of the programs will fill up meaning some students will not get their first or sometimes even their second choice.

9. controlledAllocate

   This case essentially tests for the complete correctness of the allocate function by testing every aspect of it. In this test case there are free choice students who will get their choice first, some students fail and some students may pass however will not be allocated because all three of their choice are full.

In order for all test cases to pass, many assumptions were made regarding the program's inputs and expected behavior. An example of an assumption would be that the input for department capacities and students are in a specific format in the text file or that the programmer will type input strings correctly for functions such as average.

First and foremost, I assumed that most attributes for a student are entered in the input text files correctly since many of these are computerized. For example, a lot of the information McMaster has of a student is taken from your application to McMaster, where the student's choices are entered by picking from a list or checking check boxes. This means items such as gender can't be misspelled. It also means that a list will always have most attributes of a student because in the application you can't proceed unless you've entered all the required information. Essentially the only information I assumed could be missing is if a student only made one instead of three choices for their engineering streams (because this was allowed when I was picking my streams in my first year of engineering). I made sure to check for this in my program.

I also made sure free choice students ALWAYS get their first choice since it is guaranteed to them within their admission letter. This means I must assume that there won't be more people with free choice picking a stream than there is capacity in that stream.

I also assumed that mutating variables within functions was okay because its helpful to know the current value of the variables throughout the program. For example, I mutated the list of students so that if one gets allocated I removed them from the list. This way, once the program is done going through the list we can see if any student was unaccounted for by checking if the list is empty or not.

Another assumption I made was that the average male or female gpa could not possibly be equal or less than zero or there was an error in the inputs. To avoid a division by zero error my program always checks if the gpa is zero or less before dividing. The rationale behind this is that if it is zero, then either the gender inputted is unrecognized or there were no male and/or female students in the list. In a program with 800 students its impossible that zero of them are male or zero are female. You can also assume the user wont intentionally use the function to check the gpa of a list of zero students because they would know the gpa is zero. This must mean that there is an error in the inputs, and therefore the program still runs smoothly and lets the user know that there is an error.

2 Results of Testing Partner's Code

1. sortList Passed
2. emptySortAllocate
   - Sort Passed
   - Allocate Exception Raised
3. emptyGender Exception Raised
4. wrongGender Exception Raised
5. zeroGender Exception Raised
6. AllPassAllocate Passed
7. failAllocate Sometimes Passed, Sometimes Failed
8. freeChoiceAllocate Sometimes Passed, Sometimes Failed
9. controlledAllocate Failed

3 Discussion of Test Results

After running my partner's test results, only a few passed with most receiving an exception error and two failing. The exception raises, despite being failures, are not problems with my partner's code. They're just another way of handling specific situations. For example in my test case when an empty list is passed into the average function the test case expects an output of -1. The output here can be arbitrary as long as the program knows something went wrong. In my partner's program an exception was raised which is simply another means of handling the error based off of different assumptions. For the failAllocate and freeChoiceAllocate sometimes they pass and sometimes they fail. This is because my partner shuffles free choice students so sometimes the lists outputted are in different order, however his output is always still correct. The controlledAllocate function is the only one that constantly fails and this is because of a difference in assumption where my partner gives the user a random eng stream if all their departments are full whereas I just printed everyone who wasn't allocated in a text file.

After testing my code aswell as my partners, I gained a broader insight on the excercise, and I learned how to better make assumptions and test cases in the future. For my allocate function, the controlledAllocate test case is enough to test the full correctness of the code. However, I made four different test cases so that I can better pinpoint what the problem is when an error does occur. However because my test cases sometimes incorporated more than one feature of the program it was still difficult to pinpoint what went wrong when the last two test cases failed with my partner's files. This taught me to make my test cases much more specific.

3.1 Problems with Original Code

There are a couple of problems that I saw with my code. First, I didn't have any error handling in case a student in the free choice list is not present in the students list. This could be a potential error in the inputs where perhaps the wrong file is used or something is misspelled in the free choice list. Another error with my code is that I didnt make some of my error handling outputs very specific. For example if an unrecognized gender, empty list or a list with zero of a particular gender is entered. It would be better to make it more specific so the user knows exactly whats going on.

3.2 Problems with Partner's Code

One of the design aspects that I would disagree with in my partners program is that the code stops running and raises an exception when something unexpected happens even though the program could still run. For example if a free choice student is not in the list of students an exception is raised when a print statement would suffice, and the program could keep running. I believe this will increase the reliability of the code. I also believe it would have been useful to store the list of students who failed and a list of students who didn't get any of their three choices just so the user has more background on what's happening when the program is running.

Another problem I see in my partner's code are unnecessary steps that just increase the running time of the program. For example in line 126 of CalcModule.py my partner shuffles his list and then sorts it immediately after. Another inefficient thing my partner did was store the dictionaries of all free choice students in a list and then add them to the beginning of his sorted student list. Ideally what you would want to do is store the free choice students immediately so you don't have to loop through them a second time, and even better would be to look for students who failed while already looping through the list looking for students with free choice. Even his specific implementation could have been better. In line 138 Instead of looping through EVERY student he could have just checked for the first student with below a 4.0 gpa then stopped looping since the list is sorted so you know everyone else failed.

4 Critique of Design Specification

I liked how this assignment left a lot of the design decisions open ended but at the same time left a basic structure to follow to complete the assignment. This allowed me to be more creative with the ways I handled the problems and it allowed me to solve the problem in ways that are most suited to my way of thinking.

Although this benefited me personally as it made the assignment more enjoyable overall, I would propose changing the design of the assignment to be in a more formal specification than a natural language. The ambiguity in this assignment caused a lot of assumptions to be made which might not be ideal in the real world. In the real world you'd most likely want as little ambiguity as possible that way the programmer can do exactly what is being asked.

5 Answers to Questions

(a) To make the average(L, g) function more general you can give it other capabilities specifically allowing it to take in other parameters instead of just male and female. You could have it return the gpa of students based off other input parameters (ex. have it return the average GPA of students who picked software as their first choice). In the opposite sense, you could also allow the function to output something else rather than gpa (for example the number of 'male' students who picked software as their first choice). This would require another input parameter. You can make sort(S) more general in a similar way by allowing the function to sort the list based off something other than gpa (ex. sort the list alphabetically based on the students name. This would also require another input parameter.

(b) In this context aliasing essentially means creating another name for a peice of data. For example when you set a variable equal to a dictionary it does't copy the dictionary, it just gives it another name. Therefore changing that variable will change the original dictionary. Because dictionaries are mutable then this can be a concern with dictionaries. To guard against this problem you can use the .copy method and set a variable equal to the copy of a dictionary rather than the dictionary its self.

(c) Aside from basic cases that test for the correctness of the code, I would add some test cases that test for unexpected inputs such as empty files, files with random white spaces/blank lines or files with incorrect formats. I believe CalcModule.py was selected over ReadAllocationData.py mainly because the way the files were formatted would vary greatly between students and therefore most test cases will fail with your partner's files. CalcModule is a lot more standardized with its inputs and would therefore be better to test.

(d) The problems with using strings in this way is that typing mistakes in spelling, capatalization or input type (ex int, string etc) could cause the code to break. This puts extra responsibility on the programmer to either type the strings correctly, or include error checking in their code. A better approach would be if the data structure you use helped avoid this issue. You could replace {"male", "female"} with an enumerated type, {male, female} to reduce problems. Better yet you could use {m, f} to reduce spelling errors or capitalization errors as male can be typed Male, MALE or male. A better implementation for department names would be named tuples rather than dictionary with strings as these are very readable and flexible structures and can be accessed in many different ways and therefore is less prone to error.

(e) Some of the other options of implementing the mathematical notation of a tuple is to use the built-in tuple, writing a custom class or using built-in classes. A good option for implementing a tuple in python would be to use named tuples. This would be a better choice than a dictionary with strings because its a very readable and flexible structure. Named tuples are also immutable and therefore you don't have to worry about aliasing while creating your program. It's also better than dictionaries in the sense that it makes the data passed around in your program "self-documenting" making it easier to read and follow along with whats happenning in your code. An example of how I would change my program would be to make department names a named tuple rather than a dictionary with strings. To implement this change I would have to create a named tuple class for programs with each program containing its own data. I'd then have to access each one differently than I did in my if statements by doing firstchoice.capacity for example.

(f) If I changed the list to a a different data structure then I would have to make slight changes such as indexing when trying to retreive data from the data structure. Other than that I wouldn't have to change anything else in my code because CalcModule.py for the most part is used to read data and because I'm not changing the list of choices in any way then using a tuple shouldn't be a problem. If the CalcModule.py had an ADT where the class provides a method that returns the next choice and another method that returns True when there are no more choices, if the data structure inside the custom class was changed to tuples rather than lists then assuming the same methods are available for the custom class then no you wouldn't have to change anything in CalcModule.py. Because you're just accessing the next choice and not modifying it then it should still work in the same way.

F Code for ReadAllocationData.py

```
## @file ReadAllocationData.py
#  @author Zayed Sheet
#  @brief This file contains functions that extracts student/department data from text files.
#  @date 2019-01-18


## @brief Uses the data from a text file to create a list of dictionaries.
#  @details Reads from a text file and uses the data to create a list of dictionaries where each
     dictionary represents a student.
#           \nSome of the code for this function was referenced from:\n
#
     https://stackoverflow.com/questions/4842057/easiest-way-to-ignore-blank-lines-when-reading-a-file-in-python
#  @param s This parameter is a text file that contains a student on each line. Each line should
     contain the student's macid
#           firstname, lastname, gender, gpa and top 3 engineering stream choices. \nEach line in the
     textfile should be
#           formatted as such: 'macid, firstname, lastname, gender(male/female), gpa, firstchoice,
     secondchoice, thirdchoice'.
#  @return  This function returns a list of dictionaries where each dictionary represents a student
     with their macid, fname,
#           lname, gpa, choices as keys. The choices key contains a list with their top 3 engineering
     stream choices.
def readStdnts(s):
  with open(s, "r") as f:
    #the line below creates a list of each line in the text file, ignoring empty lines
    student = [line.strip() for line in f if line.strip()]
        #https://stackoverflow.com/questions/4842057/easiest-way-to-ignore-blank-lines-when-reading-a-file-in-python
    dictionaries = []
    for i in range(len(student)):
      attributes = student[i].split(', ')
      try:
        if ((attributes[3] == 'male') | (attributes[3] == 'female')): #this ensures that the inputted
             gender is either male or female to prevent errors in average function
          dictionaries.append({
            'macid':attributes[0],
            'fname':attributes[1],
            'lname':attributes[2],
            'gender':attributes[3],
            'gpa':float(attributes[4]),
            'choices':[attributes[5],attributes[6],attributes[7]]
          })
        else:
          print("Student with macid ", attributes[0], " has been ignored due to invalid gender input")
      except IndexError:
        print("Error while processing macid ", attributes[0], " check file for missing information!")
  f.close()
  return(dictionaries)


## @brief Obtains the macids of everyone that has free choice from a text file.
#  @param s This parameter is a text file that has the macids of everyone with free choice. Each line
     should be an individual macid in the text file.
#  @return Returns a list of macids for students that have free choice.
def readFreeChoice(s):
  with open(s, "r") as f:
    macids = [line.strip() for line in f if line.strip()]
    return(macids)
  f.close()


## @brief  Obtains information on the engineering streams and their respective capacities.
#  @param s This parameter is a text file that contains the engineering streams and their respective
     capacities.
#           \nEach line in the text file should be formatted as such: 'program: capacity'
#  @return Returns a dictionary where each key is an engineering stream and the value is its'
     respective capacity.
def readDeptCapacity(s):
  departmentOptions = ['software','electrical','mechanical','engphys','chemical','civil','materials']
  with open(s, "r") as f:
    lines = [line.strip() for line in f if line.strip()]
    dictionary = {}
    for i in range(len(lines)): #for every line
      dept = lines[i].split(': ') #creates a list with department and capacity
      if ((dept[1].isdigit() == True) & (dept[0] in departmentOptions)):
        dictionary[dept[0]] = int(dept[1]) #adds onto dictionary with department as the key and
             capacity as the value
      else:
        print("Error with one of the program capacities or name!")
  f.close()
  return dictionary

# print(readStdnts("test.txt"))
# print(readFreeChoice("free.txt"))
# print(readDeptCapacity("dept.txt"))
```

G Code for CalcModule.py

```
## @file CalcModule.py
#  @author Zayed Sheet
#  @brief This file contains functions that manipulates data provided by the ReadAllocationData file.
#  @date 2019-01-18
from ReadAllocationData import *

## @brief Sorts a list of dictionaries by their 'gpa' in descending order.
#  @details This function sorts a list of dictionaries by the value of their 'gpa' key. \nSome of this
     code is referenced from
#
     \n'https://stackoverflow.com/questions/72899/how-do-i-sort-a-list-of-dictionaries-by-a-value-of-the-dictionary'
#  @param S The single parameter this function takes is a list of dictionaries, each containing a
     'gpa' key.
#  @return The returned value is a new list of dictionaries, sorted in descending order by the value
     of thier 'gpa' keys.
def sort(S):
  newlist = sorted(S, key=lambda g: g['gpa'], reverse=True)
      #https://stackoverflow.com/questions/72899/how-do-i-sort-a-list-of-dictionaries-by-a-value-of-the-dictionary
  return newlist

## @brief Calculates the average GPA of a specified gender.
#  @details The function adds together the floating point values for all dictionaries that have the
     specified 'gender' value that
#            was passed into the function. While doing this it also counts how many dictionaries
#            from the list have that gender value. It then divides the total gpa by the counter to find
     the average.
#  @param L This parameter is a list of dictionaries that have a 'gpa' and 'gender' key.
#  @param g This parameter is a string for the gender you want to find the average for.
#  @return The returned value is a value rounded to two decimal places for the average GPA of a
     specified gender.
def average(L,g):
  total = 0
  count = 0 #total number of students
  for i in L: #for every dictionary in list
   if (i['gender'] == g): #if the gender of the person is equal to given gender
     count = count + 1
     total = total + i['gpa']
  if total <= 0:
    print("Error with function input! Please check that gender is either male/female and or input is
        not empty.")
    return -1
  return(round(total/count, 2)) #return the gpa average

## @breif Allocates students into an engineering stream.
#  @details The function sorts a list of students by their GPA in descending order. It then adds
     everyone with a GPA below 4.0
#            into a text file and allocates everyone with free choice into their first choice. The
     program then allocates everyone
#            else from highest to lowest gpa into their first, second or third choice in that order
     depending on which one has
#            available capacity first. Anyone that isn't allocated into a program (because all three
     capacities are full) gets
#            placed into a text file.
#  @param S This parameter is a list of dictionaries where each dictionary represents a student with
     several attributes (such as)
#            name, macid, program choices, etc.)
#  @param F This parameter is a list of students that have free choice. Each student is represented by
     their macid.
#  @param C This parameter is a dictionary with engineering departments as keys and their respective
     capacity as the key's value.
#  @return This function returns a dictionary that contains the engineering streams as keys and a list
     of dictionaries as values.
#           Each dictionary represents a student that is allocated into that engineering stream.
def allocate(S, F, C):
  S = sort(S)
  dictionary = {'civil':[], 'chemical':[], 'electrical':[], 'mechanical':[], 'software':[],
       'materials':[], 'engphys':[]}
  failFile = open("FailList.txt", 'w') #file for everyone that failed

  i = 0
  while(i != len(S)):
    if S[i]['gpa'] < 4.0: #if students gpa is below 4
      print("student has failed: ", S[i]['macid'])
      failFile.write( #writes their student information into a text file
        "{0}, {1}, {2}, {3}, {4}, {5}, {6},
            {7}\r\n".format(S[i]['macid'],S[i]['fname'],S[i]['lname'],S[i]['gender'],S[i]['gpa'],S[i]['choices'][0],S[i]['c
      )
      del S[i] #removes that student from the list
    elif S[i]['macid'] in F: #if the student has free choice
      print("student has free choice: ", S[i]['macid'])
      dictionary[S[i]['choices'][0]].append(S[i]) #places the user into their first choice program
      C[S[i]['choices'][0]] = C[S[i]['choices'][0]] - 1 #reduces the capacity of the first choice
          program by one
      del S[i] #removes that student from the list
    else:
      i = i + 1
  failFile.close()

  unallocatedFile = open("Unallocated.txt", 'w')
  for i in range(len(S)): #loop will run for every remaining student in the list
    firstchoice = S[0]['choices'][0] #selected users first choice
    secondchoice = S[0]['choices'][1]
    thirdchoice = S[0]['choices'][2]

    if C[firstchoice] > 0: #if their first choice still has capacity (more than 0)
      dictionary[firstchoice].append(S[0]) #puts the user into their first choice program
      C[firstchoice] = C[firstchoice] - 1 #reduce the capacity by 1
      del S[0] #removes the user from the list
    elif C[secondchoice] > 0:
      dictionary[secondchoice].append(S[0])
      C[secondchoice] = C[secondchoice] - 1
      del S[0]
    elif C[thirdchoice] > 0:
      dictionary[thirdchoice].append(S[0])
      C[thirdchoice] = C[thirdchoice] - 1
      del S[0]
    else: #if the user hasnt been placed into any program because their first 3 choices were full
      print(S[0]['macid']," not allocated")
      unallocatedFile.write( #adds them into the unallocated students text file
        "{0}, {1}, {2}, {3}, {4}, {5}, {6},
            {7}\r\n".format(S[0]['macid'],S[0]['fname'],S[0]['lname'],S[0]['gender'],S[0]['gpa'],S[0]['choices'][0],S[0]['c
      )
      del S[0]
  unallocatedFile.close()
  print(len(S), " students left in the list")
  return dictionary


# test = (readStdnts("test.txt"))
# print(sort(test))
# print(average(readStdnts("test.txt"),'male'))
# print(allocate(readStdnts("test.txt"),readFreeChoice("free.txt"),readDeptCapacity("dept.txt")))
```

H Code for testCalc.py

```
## @file testCalc.py
#  @author Zayed Sheet
#  @brief This file tests the functions in the CalcModule.py file.
#  @date 2019-01-18
from CalcModule import *

emptyList = [] #this represents an empty list (empty student list, empty free choice list etc)
emptyDictionary = {}
noFreeChoice = []
unlimitedCapacity = {'engphys': 999, 'civil': 999, 'chemical': 999, 'materials': 999, 'electrical':
    999, 'mechanical': 999, 'software': 999}
controlledFreeChoice = ['sheetz', 'dominikb']
randomList = [
  {'gender': 'male', 'gpa': 10.5, 'choices': ['engphys', 'electrical', 'materials'], 'lname': 'sheet',
      'fname': 'zayed', 'macid': 'sheetz'},
  {'gender': 'male', 'gpa': 10.8, 'choices': ['software', 'electrical', 'civil'], 'lname': 'yazdinia',
      'fname': 'pedram', 'macid': 'yazdinip'},
  {'gender': 'male', 'gpa': 11.5, 'choices': ['software', 'chemical', 'electrical'], 'lname':
      'buszowiecki', 'fname': 'dominik', 'macid': 'dominikb'},
  {'gender': 'male', 'gpa': 3.2, 'choices': ['mechanical', 'electrical', 'software'], 'lname':
      'valkir', 'fname': 'farzad', 'macid': 'valkirf'},
  {'gender': 'female', 'gpa': 6.0, 'choices': ['mechanical', 'electrical', 'materials'], 'lname':
      'gonzal', 'fname': 'cat', 'macid': 'gonzalc'},
  {'gender': 'male', 'gpa': 10.9, 'choices': ['software', 'civil', 'materials'], 'lname': 'hula',
      'fname': 'mustafa', 'macid': 'mustafah'},
  {'gender': 'male', 'gpa': 10.5, 'choices': ['civil', 'electrical', 'software'], 'lname': 'samson',
      'fname': 'jordan', 'macid': 'jordans'}
]

def sortList():
  #tests the sortlist function's correctness
  #also tests if it still works when two students have the same GPA
  expectedOutput = [
    {'gender': 'male', 'gpa': 11.5, 'macid': 'dominikb', 'lname': 'buszowiecki', 'fname': 'dominik',
        'choices': ['software', 'chemical', 'electrical']},
    {'gender': 'male', 'gpa': 10.9, 'macid': 'mustafah', 'lname': 'hula', 'fname': 'mustafa',
        'choices': ['software', 'civil', 'materials']},
    {'gender': 'male', 'gpa': 10.8, 'macid': 'yazdinip', 'lname': 'yazdinia', 'fname': 'pedram',
        'choices': ['software', 'electrical', 'civil']},
    {'gender': 'male', 'gpa': 10.5, 'macid': 'sheetz', 'lname': 'sheet', 'fname': 'zayed', 'choices':
        ['engphys', 'electrical', 'materials']},
    {'gender': 'male', 'gpa': 10.5, 'macid': 'jordans', 'lname': 'samson', 'fname': 'jordan',
        'choices': ['civil', 'electrical', 'software']},
    {'gender': 'female', 'gpa': 6.0, 'macid': 'gonzalc', 'lname': 'gonzal', 'fname': 'cat', 'choices':
        ['mechanical', 'electrical', 'materials']},
    {'gender': 'male', 'gpa': 3.2, 'macid': 'valkirf', 'lname': 'valkir', 'fname': 'farzad',
        'choices': ['mechanical', 'electrical', 'software']}
  ]
  if (sort(randomList) == expectedOutput):
    print("Sort List Passed!")
  else:
    print("Sort List Failed :(")

def emptySortAllocate():
  #if an empty list is passed into the alllocate function
  #first sort is checked because if sort doesnt work allocate wont work anyways
  #if sort works then it checks allocate with an empty list in several different cases
  emptyCapacityDictionary = {'engphys': [], 'civil': [], 'chemical': [], 'materials': [],
      'electrical': [], 'mechanical': [], 'software': []}

  if (sort(emptyList) == []):
    print("Empty Sort Passed!")
    if ((allocate(emptyList,emptyList,emptyDictionary) == emptyCapacityDictionary) &
        (allocate(emptyList,emptyList,unlimitedCapacity) == emptyCapacityDictionary) &
        (allocate(emptyList,controlledFreeChoice,emptyDictionary) == emptyCapacityDictionary)):
      print("Empty Allocation Passed!")
    else:
      print("Empty Allocation Failed :(")
  else:
    print("Empty Sort Failed :(")

def emptyGender():
  #if an empty list is passed into the average function
  if (average(emptyList, 'male') == -1):
    print("Empty Gender Passed!")
  else:
    print("Empty Gender Failed :(")

def wrongGender():
  #if the programer does not enter either male or female as a parameter
  if (average(randomList, 'test') == -1):
    print("Wrong Gender Passed!")
  else:
    print("Wrong Gender Failed :(")

def zeroGender():
  #if the programer enters a list with zero males or zero females as a parameter
  zeroMaleList = [
    {'gender': 'female', 'gpa': 10.5, 'choices': ['engphys', 'electrical', 'materials'], 'lname':
        'sheet', 'fname': 'zayed', 'macid': 'sheetz'},
    {'gender': 'female', 'gpa': 10.8, 'choices': ['software', 'electrical', 'civil'], 'lname':
        'yazdinia', 'fname': 'pedram', 'macid': 'yazdinip'}
  ]
  if ((average(zeroMaleList, 'male') == -1)):
    print("Zero Gender Passed!")
  else:
    print("Zero Gender Failed :(")


def allPassAllocate():
  #In this test case everyone passes, theres unlimited capacity, no free choice students and everyone
      gets their first choice
  allPassList = [
    {'gender': 'male', 'gpa': 10.5, 'choices': ['software', 'electrical', 'materials'], 'lname':
        'sheet', 'fname': 'zayed', 'macid': 'sheetz'},
    {'gender': 'male', 'gpa': 10.8, 'choices': ['chemical', 'electrical', 'civil'], 'lname':
        'yazdinia', 'fname': 'pedram', 'macid': 'yazdinip'},
    {'gender': 'male', 'gpa': 11.5, 'choices': ['software', 'chemical', 'electrical'], 'lname':
        'buszowiecki', 'fname': 'dominik', 'macid': 'dominikb'},
    {'gender': 'male', 'gpa': 7.2, 'choices': ['materials', 'electrical', 'software'], 'lname':
        'valkir', 'fname': 'farzad', 'macid': 'valkirf'},
    {'gender': 'female', 'gpa': 6.0, 'choices': ['mechanical', 'electrical', 'materials'], 'lname':
        'gonzal', 'fname': 'cat', 'macid': 'gonzalc'},
    {'gender': 'male', 'gpa': 10.9, 'choices': ['civil', 'mechanical', 'materials'], 'lname': 'hula',
        'fname': 'mustafa', 'macid': 'mustafah'},
    {'gender': 'male', 'gpa': 10.2, 'choices': ['engphys', 'electrical', 'software'], 'lname':
        'samson', 'fname': 'jordan', 'macid': 'jordans'}
  ]
  expectedOutput = {
    'engphys': [{'gender': 'male', 'gpa': 10.2, 'choices': ['engphys', 'electrical', 'software'],
        'lname': 'samson', 'fname': 'jordan', 'macid': 'jordans'}],
    'civil': [{'gender': 'male', 'gpa': 10.9, 'choices': ['civil', 'mechanical', 'materials'],'lname':
        'hula', 'fname': 'mustafa', 'macid': 'mustafah'}],
    'chemical': [{'gender': 'male', 'gpa': 10.8, 'choices': ['chemical', 'electrical', 'civil'],
        'lname': 'yazdinia', 'fname': 'pedram', 'macid': 'yazdinip'}],
    'materials': [{'gender': 'male', 'gpa': 7.2, 'choices': ['materials', 'electrical', 'software'],
        'lname': 'valkir', 'fname': 'farzad', 'macid': 'valkirf'}],
    'electrical': [],
    'mechanical': [{'gender': 'female', 'gpa': 6.0, 'choices': ['mechanical', 'electrical',
        'materials'], 'lname': 'gonzal', 'fname': 'cat', 'macid': 'gonzalc'}],
    'software': [{'gender': 'male', 'gpa': 11.5, 'choices': ['software', 'chemical', 'electrical'],
        'lname': 'buszowiecki', 'fname': 'dominik', 'macid': 'dominikb'},{'gender': 'male', 'gpa':
        10.5, 'choices': ['software', 'electrical', 'materials'], 'lname': 'sheet', 'fname': 'zayed',
        'macid': 'sheetz'}]
  }
  if (allocate(allPassList,noFreeChoice,unlimitedCapacity) == expectedOutput):
    print("All Pass Allocate Passed!")
  else:
    print("All Pass Allocate Failed :(")

def failAllocate():
  #In this test case a few students fail, they should not be on the expectedOutput
  failList = [
    {'gender': 'male', 'gpa': 10.5, 'choices': ['software', 'electrical', 'materials'], 'lname':
        'sheet', 'fname': 'zayed', 'macid': 'sheetz'},
    {'gender': 'male', 'gpa': 10.8, 'choices': ['chemical', 'electrical', 'civil'], 'lname':
        'yazdinia', 'fname': 'pedram', 'macid': 'yazdinip'},
    {'gender': 'male', 'gpa': 11.5, 'choices': ['software', 'chemical', 'electrical'], 'lname':
        'buszowiecki', 'fname': 'dominik', 'macid': 'dominikb'},
    {'gender': 'male', 'gpa': 2.2, 'choices': ['materials', 'electrical', 'software'], 'lname':
        'valkir', 'fname': 'farzad', 'macid': 'valkirf'},
    {'gender': 'female', 'gpa': 6.0, 'choices': ['mechanical', 'electrical', 'materials'], 'lname':
        'gonzal', 'fname': 'cat', 'macid': 'gonzalc'},
    {'gender': 'male', 'gpa': 0, 'choices': ['civil', 'mechanical', 'materials'], 'lname': 'hula',
        'fname': 'mustafa', 'macid': 'mustafah'},
    {'gender': 'male', 'gpa': 10.2, 'choices': ['civil', 'engphys', 'software'], 'lname': 'samson',
        'fname': 'jordan', 'macid': 'jordans'}
  ]
  expectedOutput = {
    'engphys': [],
    'civil': [{'gender': 'male', 'gpa': 10.2, 'choices': ['civil', 'engphys', 'software'], 'lname':
        'samson', 'fname': 'jordan', 'macid': 'jordans'}],
    'chemical': [{'gender': 'male', 'gpa': 10.8, 'choices': ['chemical', 'electrical', 'civil'],
        'lname': 'yazdinia', 'fname': 'pedram', 'macid': 'yazdinip'}],
    'materials': [],
    'electrical': [],
    'mechanical': [{'gender': 'female', 'gpa': 6.0, 'choices': ['mechanical', 'electrical',
        'materials'], 'lname': 'gonzal', 'fname': 'cat', 'macid': 'gonzalc'}],
    'software': [{'gender': 'male', 'gpa': 11.5, 'choices': ['software', 'chemical', 'electrical'],
        'lname': 'buszowiecki', 'fname': 'dominik', 'macid': 'dominikb'},{'gender': 'male', 'gpa':
        10.5, 'choices': ['software', 'electrical', 'materials'], 'lname': 'sheet', 'fname': 'zayed',
        'macid': 'sheetz'}]
  }
  if (allocate(failList,controlledFreeChoice,unlimitedCapacity) == expectedOutput):
    print("Fail Allocate Passed!")
  else:
    print("Fail Allocate Failed :(")



def freeChoiceAllocate():
  #In this test case  capacity is limited to ensure free choice students get the program they desire
      first.
  #In turn it also technically checks what happens if a student's choice is full.
  freeChoiceList = [
    {'gender': 'male', 'gpa': 10.5, 'choices': ['software', 'electrical', 'materials'], 'lname':
        'sheet', 'fname': 'zayed', 'macid': 'sheetz'},
    {'gender': 'male', 'gpa': 10.8, 'choices': ['chemical', 'electrical', 'civil'], 'lname':
        'yazdinia', 'fname': 'pedram', 'macid': 'yazdinip'},
    {'gender': 'male', 'gpa': 11.5, 'choices': ['software', 'chemical', 'electrical'], 'lname':
        'buszowiecki', 'fname': 'dominik', 'macid': 'dominikb'},
    {'gender': 'male', 'gpa': 4.2, 'choices': ['electrical', 'materials', 'software'], 'lname':
        'valkir', 'fname': 'farzad', 'macid': 'valkirf'},
    {'gender': 'female', 'gpa': 6.0, 'choices': ['mechanical', 'electrical', 'materials'], 'lname':
        'gonzal', 'fname': 'cat', 'macid': 'gonzalc'},
    {'gender': 'male', 'gpa': 4.0, 'choices': ['civil', 'mechanical', 'materials'], 'lname': 'hula',
        'fname': 'mustafa', 'macid': 'mustafah'},
    {'gender': 'male', 'gpa': 10.2, 'choices': ['software', 'engphys', 'electrical'], 'lname':
        'samson', 'fname': 'jordan', 'macid': 'jordans'}
  ]
  freeCapacity = {'engphys': 0, 'civil': 999, 'chemical': 999, 'materials': 999, 'electrical': 1,
      'mechanical': 999, 'software': 2}
  expectedOutput = {
    'engphys': [],
    'civil': [{'gender': 'male', 'gpa': 4.0, 'choices': ['civil', 'mechanical', 'materials'], 'lname':
        'hula', 'fname': 'mustafa', 'macid': 'mustafah'}],
    'chemical': [{'gender': 'male', 'gpa': 10.8, 'choices': ['chemical', 'electrical', 'civil'],
        'lname': 'yazdinia', 'fname': 'pedram', 'macid': 'yazdinip'}],
    'materials': [{'gender': 'male', 'gpa': 4.2, 'choices': ['electrical', 'materials', 'software'],
        'lname': 'valkir', 'fname': 'farzad', 'macid': 'valkirf'}],
    'electrical': [{'gender': 'male', 'gpa': 10.2, 'choices': ['software', 'engphys', 'electrical'],
        'lname': 'samson', 'fname': 'jordan', 'macid': 'jordans'}],
    'mechanical': [{'gender': 'female', 'gpa': 6.0, 'choices': ['mechanical', 'electrical',
        'materials'], 'lname': 'gonzal', 'fname': 'cat', 'macid': 'gonzalc'}],
    'software': [{'gender': 'male', 'gpa': 11.5, 'choices': ['software', 'chemical', 'electrical'],
        'lname': 'buszowiecki', 'fname': 'dominik', 'macid': 'dominikb'},{'gender': 'male', 'gpa':
        10.5, 'choices': ['software', 'electrical', 'materials'], 'lname': 'sheet', 'fname': 'zayed',
        'macid': 'sheetz'}]
  }
  if (allocate(freeChoiceList,controlledFreeChoice,freeCapacity) == expectedOutput):
    print("Free Choice Allocate Passed!")
  else:
    print("Free Choice Allocate Failed :(")

def controlledAllocate():
  #in this test case there will be some students who failed, some who have free choice and some
      students that passed however all three of their choices will be filled up
  controlledList = [
    {'gender': 'male', 'gpa': 10.5, 'choices': ['software', 'electrical', 'materials'], 'lname':
        'sheet', 'fname': 'zayed', 'macid': 'sheetz'},
    {'gender': 'male', 'gpa': 10.8, 'choices': ['software', 'electrical', 'civil'], 'lname':
        'yazdinia', 'fname': 'pedram', 'macid': 'yazdinip'},
    {'gender': 'male', 'gpa': 11.5, 'choices': ['software', 'chemical', 'electrical'], 'lname':
        'buszowiecki', 'fname': 'dominik', 'macid': 'dominikb'},
    {'gender': 'male', 'gpa': 3.2, 'choices': ['mechanical', 'electrical', 'software'], 'lname':
         'valkir', 'fname': 'farzad', 'macid': 'valkirf'},
    {'gender': 'female', 'gpa': 6.0, 'choices': ['mechanical', 'electrical', 'materials'], 'lname':
         'gonzal', 'fname': 'cat', 'macid': 'gonzalc'},
    {'gender': 'male', 'gpa': 10.9, 'choices': ['software', 'civil', 'materials'], 'lname': 'hula',
         'fname': 'mustafa', 'macid': 'mustafah'},
    {'gender': 'male', 'gpa': 10.2, 'choices': ['civil', 'electrical', 'software'], 'lname': 'samson',
         'fname': 'jordan', 'macid': 'jordans'}
  ]
  controlledCapacity = {'engphys': 3, 'civil': 1, 'chemical': 2, 'materials': 0, 'electrical': 0,
       'mechanical': 3, 'software': 2}
  expectedOutput = {
    'engphys': [],
    'civil': [{'gender': 'male', 'gpa': 10.9, 'choices': ['software', 'civil', 'materials'],
    'lname': 'hula', 'fname': 'mustafa', 'macid': 'mustafah'}],
    'chemical': [],
    'materials': [],
    'electrical': [],
    'mechanical': [{'gender': 'female', 'gpa': 6.0, 'choices': ['mechanical', 'electrical',
         'materials'], 'lname': 'gonzal', 'fname': 'cat', 'macid': 'gonzalc'}],
    'software': [{'gender': 'male', 'gpa': 11.5, 'choices': ['software', 'chemical', 'electrical'],
         'lname': 'buszowiecki', 'fname': 'dominik', 'macid': 'dominikb'},{'gender': 'male', 'gpa':
         10.5, 'choices': ['software', 'electrical', 'materials'], 'lname': 'sheet', 'fname': 'zayed',
         'macid': 'sheetz'}]
  }

  if (allocate(controlledList,controlledFreeChoice,controlledCapacity) == expectedOutput):
    print("Controlled Allocate Passed!")
  else:
    print("Controlled Allocate Failed :(")



sortList()
emptySortAllocate()
emptyGender()
wrongGender()
zeroGender()
allPassAllocate()
failAllocate()
freeChoiceAllocate()
controlledAllocate()
```

I Code for Partner's CalcModule.py

```
## @file CalcModule.py
#  @author Jack Buckley
#  @brief A module containing the core functions used for allocating a student to their program.
#  @date 18 January 2019

from statistics import mean
from random import shuffle
import Exceptions

## @brief Sorts a list of student dictionaries by GPA.
#  @details The student dictionaries are sorted in descending order in terms of GPA. It is assumed that
#           the dictionary for each student will not be changed (won't cause aliasing issues).
#  @param S Takes in list of student dictionaries of the form generated by readStdnts(s)
#  @return A list of student dictionaries sorted by GPA.

# Credit on how to sort lists in Python using a key function:
# https://www.geeksforgeeks.org/python-list-sort/

def sort(S):
    # Make a copy of S (otherwise side effects will arise using list.sort())
    D = S.copy()
    # A lambda function which returns the value of a dictionary at the key 'gpa'
    sortByGPA = lambda diction : diction['gpa']

    # Sort the list of dictionary S by the GPA of each student in descending orders
    D.sort(key = sortByGPA, reverse = True)
    return D

## @brief Finds the average GPA of all males or females.
#  @param L Takes in list of student dictionaries of the form generated by readStdnts(s)
#  @param g Takes in string of gender to find the average of: "male" or "female"
#  @exception GenderNotGiven raised if g is not "male" or "female"
#  @return A float representing the average GPA of the specified gender

def average(L, g):
    # A list where male GPAs will go
    maleGPAs = []
    # A list where female GPAs will go
    femaleGPAs = []

    # Iterate through each student dictionary
    for studentDict in L:
        # Get the gender of the current student
        gender = studentDict['gender']

        # If the current student is male, add his GPA to the male GPA list
        if(gender == 'male'):
            maleGPAs += [studentDict['gpa']]

        # If the current student is female, add her GPA to the male GPA list
        elif(gender == 'female'):
            femaleGPAs += [studentDict['gpa']]

    # If the user wants to know the male GPA, find the mean of the male GPA list
    if(g == 'male'):
        return mean(maleGPAs)

    # If the user wants to know the female GPA, find the mean of the male GPA list
    elif(g == 'female'):
        return mean(femaleGPAs)
    else:
        raise Exceptions.GenderNotGiven('g is not "male" or "female"')

## @brief Allocates students to their programs.
#  @details Students who have free choice get first pick, followed by the remaining students according
    to their GPA
#           with those with higher GPAs getting higher priority. Those eligible will be also shuffled
    in a lottery
#           according to their eligibility. That is to say all students with free choice will be
    shuffled among
#           themselves and all students with the same GPA will be shuffled among themselves as well.
    Any student,
#           regardless of if they have free choice, who has a GPA of less than 4.0 will not be
    allocated to a program.
#           While students are allocated in the aforementioned order, if a department no longer has
    capacity for additional
```

```
#           students, students will allocated to their second/third preferences. If it arises that all
    three of a student's
#           choices are at full capacity, the student will be assigned to a random department with
    room.
#
#  @param S Takes in list of student dictionaries of the form generated by readStdnts(s)
#  @param F Takes in list of students' MacID strings for those who have free choice in the form
    generated by readFreeChoice(s)
#  @param C Takes in dictionary of department capacities of the form generated by readDeptCapacity(s)
#  @return A dictionary where the department names are the keys and their respective values are lists
    of student dictonaries of
#          students who have been allocated to them.
#  @exception Exceptions.FreeChoiceStudentNotAccountedFor when a student in the free choice list
    cannot be allocated to a program
#              because their student dictionary is not in the dictionary list.
#  @exception Exceptions.NoDepartmentsHaveCapacity when no departments have capacity but there still
    exist students to be allocated
#              to programs.
#  @exception Exceptions.EmptyStudentDictionaryList when parameter S containing a list of student
    dictionaries is empty.
#  @exception Exceptions.EmptyDepartmentCapacityDict when parameter C containing a dictionary of
    department capacities is empty.

def allocate(S, F, C):
    # Do not want to mutate inputs, so copying them to prevent side effects
    SLocal = S.copy()
    FLocal = F.copy()
    CLocal = C.copy()


    # We should not allocate any students if the student dictionary list or department
    # capacity dictionary are empty (It's OK if free choice is empty, however)
    if SLocal == []:
        raise Exceptions.EmptyStudentDictionaryList

    elif CLocal == {}:
        raise Exceptions.EmptyDepartmentCapacityDict


    # The allocation dictionary
    allocation = {'civil': [], 'chemical': [], 'electrical': [], 'mechanical': [], 'software': [],
        'materials': [], 'engphys': []}


    # Get the student dictonaries for those with free choice from the list SLocal and remove them from
        SLocal
    freeChoiceDicts = []
    for freeChoiceStudent in FLocal:
        # Try to get the student's information from the list of dictionaries SLocal by matching
            against their MacID
        try:
            stdntInfo = list(filter(lambda diction : diction["macid"] == freeChoiceStudent, SLocal))[0]
        except:
            raise Exceptions.FreeChoiceStudentNotAccountedFor("A student on the free choice list is
                not" +
            "in the list of student dictionaries and thus cannot be allocated.")

        # Remove free choice student from the list of student dictionaries
        SLocal.remove(stdntInfo)

        # Place their dictionary in the list of dictionaries of those with free choice
        freeChoiceDicts += [stdntInfo]

    # Shuffle those with free choice
    shuffle(freeChoiceDicts)
    # Shuffle everyone else
    shuffle(SLocal)


    # Sort everyone else by GPA
    SLocal = sort(SLocal)
    # print(SLocal)
    # print("\n")

    # Combine the student dictonaries so first the students with free choice get first pick
    # Then, everyone else gets to pick starting with those with the highest GPA
    SLocal = freeChoiceDicts + SLocal

    # Iterate through each student's dictonary in the list.
    # The list now has the students with the highest priority at the beginning of the list and
    # accounts for those with free choice
    for student in SLocal:
        gpa = student["gpa"]
        # If the student has less than a 4.0 GPA, they will not be allocated
        if (gpa < 4.0):
            continue

        # Extract the student's choices from their dictionary
        choices = student["choices"]
        fstChoice = choices[0]
        sndChoice = choices[1]
        thrdChoice = choices[2]

        # If the first choice department has spots available, put the student in that program
        if(CLocal[fstChoice] != 0):
            allocation[fstChoice] += [student]

            # Decrease capacity by 1
            CLocal[fstChoice] -= 1

        # If not, check if there are spots in the student's second preference and place them there if
            there is
        elif(CLocal[sndChoice] != 0):
            allocation[sndChoice] += [student]

            # Decrease capacity by 1
            CLocal[sndChoice] -= 1

        # If not, check if there are spots in the student's third preference and place them there if
            there is
        elif(CLocal[thrdChoice] != 0):
            allocation[thrdChoice] += [student]

            # Decrease capacity by 1
            CLocal[thrdChoice] -= 1

        # Otherwise, put the student in a random department that still has capacity
        else:
            # Get all the departments
            depts = list(CLocal.keys())
            deptsWithRoom = []

            # Add to the departments with room list
            for dept in depts:
                if CLocal[dept] != 0:
                    deptsWithRoom += [dept]

            # If deptsWithRoom is still empty, there is not enough room for students
            if deptsWithRoom == []:
                raise Exceptions.NoDepartmentsHaveCapacity

            # Shuffle the departments with room
            shuffle(deptsWithRoom)

            # The student is assigned a random department with room
            assignedDept = deptsWithRoom[0]
            allocation[assignedDept] += [student]

            # Decrease capacity by 1
            CLocal[assignedDept] -= 1
    print(allocation)
    return allocation
```

J Makefile

```
\documentclass[12pt]{article}

\usepackage{graphicx}
\usepackage{paralist}
\usepackage{listings}
\usepackage{booktabs}
\usepackage{indentfirst}

\oddsidemargin 0mm
\evensidemargin 0mm
\textwidth 160mm
\textheight 200mm

\pagestyle {plain}
\pagenumbering{arabic}

\newcounter{stepnum}

\title{Assignment 1 Report}
\author{Zayed Sheet, Sheetz}
\date{\today}

\begin {document}

\maketitle

The following document is a thorough report on the 2AA4 Assignment 1. The purpose of this software
     design exercise is to write a python program that uses three modules named ReadAllocationData.py,
     CalcModule.py and testCalc.py to take an input of students in first year engineering and place
     them in one of their desired programs of choice depending on whether they meet the requirements
     for that program.

\section{Testing of the Original Program}

When creating my test cases, the first thing I did was try to cover all the basic cases that
     essentially assess the correctness of the code.
These basic cases would have expected inputs and expected outputs to test the basic functionality of
     the code.
I then added some cases that test the robustness of the program by essentially trying to break the
     code with unexpected inputs.
For example, cases where the inputs would contain empty lists, empty dictionaries or unrecognized
     string literals. My test cases go as follows:

\bigskip
While developing my test cases, the only issue I ran into was an unexpected output due to one of my
     global variables being changed in another test case.
Other than that every test case passed, presumably because I was testing my code as I was writing it,
     and because my insight didn't really change much while writing my test cases, since writing my
     code. My test cases go as follows:

\begin{enumerate}
   \item sortList

The purpose of this test case is to test the basic functionality of the sort function. A list of
     dictionaries is passed into the function and the expected output is a list of dictionaries sorted
     by the 'gpa' key in descending order.
   \item emptySortAllocate

This test case tests the robustness of the program when empty parameters are passed in. There are two
     parts to this test case. First it checks if the sort function works when an empty list is passed
     in. The expected output for this is an empty list. If this works then it checks if the allocate
     function works when the parameters are empty. Because the allocate function takes several
     parameters, the test case checks three different combinations of parameters to see if it still
     works when one parameter is empty but others are not. The reason I combined two test cases in
     this function is because if the sort case fails, then the allocate function will fail anyways.
   \item emptyGender

This test case tests what happens if an empty list is passed into the average function. The rationale
     being that the user may have entered the wrong list as an input but the program should still run
     without error.
   \item wrongGender

This test case tests how the program handles unexpected inputs. In this test case the gender parameter
     into the average function is 'test' rather than 'male' or 'female'. The rationale behind this is
     in-case the programer or user mistypes male or female or enters any other input for the gender
     parameter. The expected output is -1.
   \item zeroGender
```

```
The purpose for this test case is to see what happens when the list of students doesn't have a single
     'male' and/or 'female'. For example, if all the students were male but female is passed into the
     function to test for the average gpa. The rationale behind this test case is to ensure there is
     no zero by division error despite there being a reasonable input. The expected output is −1.
   \item allPassAllocate

This is one of many test cases to test the basic functionality of the allocate function. I created
     several test cases for allocate to pinpoint where the problem is if an error did occur. This as
     well as the rest of my test cases are to test for correctness because the allocate function has
     several aspects to it. In this test case all students pass and are allocated to their first
     choice. There are no capacities or free choice students.
   \item failAllocate

In this test case some students fail, and therefore should not be present in the list outputted by the
     function. This test case tests if the program still works when students fail. There are no free
     choice students, no capacity and everyone gets their first choice if they passed.
   \item freeChoiceAllocate

In this test there are free choice students, all students are allocated, capacities are limited
     however all students pass. Free choice students should always get their first choice and should
     be entered into their program before other students. Subsequently, some of the programs will fill
     up meaning some students will not get their first or sometimes even their second choice.
   \item controlledAllocate

This case essentially tests for the complete correctness of the allocate function by testing every
     aspect of it. In this test case there are free choice students who will get their choice first,
     some students fail and some students may pass however will not be allocated because all three of
     their choice are full.
\end{enumerate}

In order for all test cases to pass, many assumptions were made regarding the program's inputs and
     expected behavior. An example of an assumption would be that the input for department capacities
     and students are in a specific format in the text file or that the programmer will type input
     strings correctly for functions such as average.

\bigskip
First and foremost, I assumed that most attributes for a student are entered in the input text files
     correctly since many of these are computerized. For example, a lot of the information McMaster
     has of a student is taken from your application to McMaster, where the student's choices are
     entered by picking from a list or checking check boxes. This means items such as gender can't be
     misspelled. It also means that a list will always have most attributes of a student because in
     the application you can't proceed unless you've entered all the required information. Essentially
     the only information I assumed could be missing is if a student only made one instead of three
     choices for their engineering streams (because this was allowed when I was picking my streams in
     my first year of engineering). I made sure to check for this in my program.

\bigskip
I also made sure free choice students ALWAYS get their first choice since it is guaranteed to them
     within their admission letter. This means I must assume that there won't be more people with free
     choice picking a stream than there is capacity in that stream.

\bigskip
I also assumed that mutating variables within functions was okay because its helpful to know the
     current value of the variables throughout the program. For example, I mutated the list of
     students so that if one gets allocated I removed them from the list. This way, once the program
     is done going through the list we can see if any student was unaccounted for by checking if the
     list is empty or not.

\bigskip
Another assumption I made was that the average male or female gpa could not possibly be equal or less
     than zero or there was an error in the inputs. To avoid a division by zero error my program
     always checks if the gpa is zero or less before dividing. The rationale behind this is that if it
     is zero, then either the gender inputted is unrecognized or there were no male and/or female
     students in the list. In a program with 800 students its impossible that zero of them are male or
     zero are female. You can also assume the user wont intentionally use the function to check the
     gpa of a list of zero students because they would know the gpa is zero. This must mean that there
     is an error in the inputs, and therefore the program still runs smoothly and lets the user know
     that there is an error.



\section{Results of Testing Partner's Code}
\begin{enumerate}
   \item sortList Passed
   \item emptySortAllocate
   \begin{itemize}
     \item Sort Passed
     \item Allocate Exception Raised
   \end{itemize}
```

```
    \item emptyGender Exception Raised
    \item wrongGender Exception Raised
    \item zeroGender Exception Raised
    \item AllPassAllocate Passed
    \item failAllocate Sometimes Passed, Sometimes Failed
    \item freeChoiceAllocate Sometimes Passed, Sometimes Failed
    \item controlledAllocate Failed
\end{enumerate}


\section{Discussion of Test Results}

After running my partner's test results, only a few passed with most receiving an exception error and
     two failing. The exception raises, despite being failures, are not problems with my partner's
     code. They're just another way of handling specific situations. For example in my test case when
     an empty list is passed into the average function the test case expects an output of -1. The
     output here can be arbitrary as long as the program knows something went wrong. In my partner's
     program an exception was raised which is simply another means of handling the error based off of
     different assumptions. For the failAllocate and freeChoiceAllocate sometimes they pass and
     sometimes they fail. This is because my partner shuffles free choice students so sometimes the
     lists outputted are in different order, however his output is always still correct. The
     controlledAllocate function is the only one that constantly fails and this is because of a
     difference in assumption where my partner gives the user a random eng stream if all their
     departments are full whereas I just printed everyone who wasn't allocated in a text file.

After testing my code aswell as my partners, I gained a broader insight on the excercise, and I
     learned how to better make assumptions and test cases in the future. For my allocate function,
     the controlledAllocate test case is enough to test the full correctness of the code. However, I
     made four different test cases so that I can better pinpoint what the problem is when an error
     does occur. However because my test cases sometimes incorporated more than one feature of the
     program it was still difficult to pinpoint what went wrong when the last two test cases failed
     with my partner's files. This taught me to make my test cases much more specific.

\subsection{Problems with Original Code}

There are a couple of problems that I saw with my code. First, I didn't have any error handling in
     case a student in the free choice list is not present in the students list. This could be a
     potential error in the inputs where perhaps the wrong file is used or something is misspelled in
     the free choice list. Another error with my code is that I didnt make some of my error handling
     outputs very specific. For example if an unrecognized gender, empty list or a list with zero of a
     particular gender is entered. It would be better to make it more specific so the user knows
     exactly whats going on.

\subsection{Problems with Partner's Code}

One of the design aspects that I would disagree with in my partners program is that the code stops
     running and raises an exception when something unexpected happens even though the program could
     still run. For example if a free choice student is not in the list of students an exception is
     raised when a print statement would suffice, and the program could keep running. I believe this
     will increase the reliability of the code. I also believe it would have been useful to store the
     list of students who failed and a list of students who didn't get any of their three choices just
     so the user has more background on what's happening when the program is running.

Another problem I see in my partner's code are unnecessary steps that just increase the running time
     of the program. For example in line 126 of CalcModule.py my partner shuffles his list and then
     sorts it immediately after. Another inefficient thing my partner did was store the dictionaries
     of all free choice students in a list and then add them to the beginning of his sorted student
     list. Ideally what you would want to do is store the free choice students immediately so you
     don't have to loop through them a second time, and even better would be to look for students who
     failed while already looping through the list looking for students with free choice. Even his
     specific implementation could have been better. In line 138 Instead of looping through EVERY
     student he could have just checked for the first student with below a 4.0 gpa then stopped
     looping since the list is sorted so you know everyone else failed.
\section{Critique of Design Specification}

I liked how this assignment left a lot of the design decisions open ended but at the same time left a
     basic structure to follow to complete the assignment. This allowed me to be more creative with
     the ways I handled the problems and it allowed me to solve the problem in ways that are most
     suited to my way of thinking.

Although this benefited me personally as it made the assignment more enjoyable overall, I would
     propose changing the design of the assignment to be in a more formal specification than a natural
     language. The ambiguity in this assignment caused a lot of assumptions to be made which might not
     be ideal in the real world. In the real world you'd most likely want as little ambiguity as
     possible that way the programmer can do exactly what is being asked.
%\newpage

\section{Answers to Questions}

\begin{enumerate}[(a)]
```

```
\item To make the average(L, g) function more general you can give it other capabilities specifically
     allowing it to take in other parameters instead of just male and female. You could have it return
     the gpa of students based off other input parameters (ex. have it return the average GPA of
     students who picked software as their first choice). In the opposite sense, you could also allow
     the function to output something else rather than gpa (for example the number of 'male' students
     who picked software as their first choice). This would require another input parameter. You can
     make sort(S) more general in a similar way by allowing the function to sort the list based off
     something other than gpa (ex. sort the list alphabetically based on the students name. This would
     also require another input parameter.

\item In this context aliasing essentially means creating another name for a peice of data. For
     example when you set a variable equal to a dictionary it does't copy the dictionary, it just
     gives it another name. Therefore changing that variable will change the original dictionary.
     Because dictionaries are mutable then this can be a concern with dictionaries. To guard against
     this problem you can use the .copy method and set a variable equal to the copy of a dictionary
     rather than the dictionary its self.

\item Aside from basic cases that test for the correctness of the code, I would add some test cases
     that test for unexpected inputs such as empty files, files with random white spaces/blank lines
     or files with incorrect formats. I believe CalcModule.py was selected over ReadAllocationData.py
     mainly because the way the files were formatted would vary greatly between students and therefore
     most test cases will fail with your partner's files. CalcModule is a lot more standardized with
     its inputs and would therefore be better to test.

\item The problems with using strings in this way is that typing mistakes in spelling, capatalization
     or input type (ex int, string etc)  could cause the code to break. This puts extra responsibility
     on the programmer to either type the strings correctly, or include error checking in their code.
     A better approach would be if the data structure you use helped avoid this issue. You could
     replace \{"male", "female"\} with an enumerated type, \{male, female\} to reduce problems. Better
     yet you could use \{m, f\} to reduce spelling errors or capitalization errors as male can be
     typed Male, MALE or male. A better implementation for department names would be named tuples
     rather than dictionary with strings as these are very readable and flexible structures and can be
     accessed in many different ways and therefore is less prone to error.

\item Some of the other options of implementing the mathematical notation of a tuple is to use the
     built-in tuple, writing a custom class or using built-in classes. A good option for implementing
     a tuple in python would be to use named tuples. This would be a better choice than a dictionary
     with strings because its a very readable and flexible structure. Named tuples are also immutable
     and therefore you don't have to worry about aliasing while creating your program. It's also
     better than dictionaries in the sense that it makes the data passed around in your program
     "self-documenting" making it easier to read and follow along with whats happenning in your code.
     An example of how I would change my program would be to make department names a named tuple
     rather than a dictionary with strings. To implement this change I would have to create a named
     tuple class for programs with each program containing its own data. I'd then have to access each
     one differently than I did in my if statements by doing firstchoice.capacity for example.

\item If I changed the list to a a different data structure then I would have to make slight changes
     such as indexing when trying to retreive data from the data structure. Other than that I wouldn't
     have to change anything else in my code because CalcModule.py for the most part is used to read
     data and because I'm not changing the list of choices in any way then using a tuple shouldn't be
     a problem. If the CalcModule.py had an ADT where the class provides a method that returns the
     next choice and another method that returns True when there are no more choices, if the data
     structure inside the custom class was changed to tuples rather than lists then assuming the same
     methods are available for the custom class then no you wouldn't have to change anything in
     CalcModule.py. Because you're just accessing the next choice and not modifying it then it should
     still work in the same way.
\end{enumerate}

\newpage

\lstset{language=Python, basicstyle=\tiny, breaklines=true, showspaces=false,
  showstringspaces=false, breakatwhitespace=true}
%\lstset{language=C,linewidth=.94\textwidth,xleftmargin=1.1cm}

\def\thesection{\Alph{section}}

\section{Code for ReadAllocationData.py}

\noindent \lstinputlisting{../src/ReadAllocationData.py}

\newpage

\section{Code for CalcModule.py}

\noindent \lstinputlisting{../src/CalcModule.py}

\newpage

\section{Code for testCalc.py}
```

```
\noindent \lstinputlisting{../src/testCalc.py}

\newpage

\section{Code for Partner's CalcModule.py}

\noindent \lstinputlisting{../partner/CalcModule.py}

\newpage

\section{Makefile}

\lstset{language=make}
\noindent \lstinputlisting{../Makefile}

\end {document}
```