Assignment 1

Bill Nguyen and nguyew3

January 24, 2019

For this report what I will be discussing are to start the testing process, in which I explain each test cases and results of each test for my program and also my partner's program. Also I will list assumptions I made and my opinion about the design specification of the project. Finally I will be answering the questions mentioned in step 8.

1 Testing of the Original Program

For testing my approach was to have cases that were expected, cases that were normal but less likely to happen and cases that were extreme or due to human error. This was my reasoning in order to make sure program works in many different scenarios. All test cases below, my program passed all these cases.

- testSort1 - test when there is no students, result should return an empty list. Decided to use this as an extreme case to see what happens if there is no info to sort.
- testSort2 - test with inputs you would expect, with no input error. Made this a normal case too see if my implementation actually works.
- testAverage1 - test when there is no students and should return 0. The reason I decided to test is the same reason as testSort1 and also made it return 0 to avoid zerobydivison error.
- testAverage2 - test with expected inputs to see if implementation is correct
- testAverage3 - test with only male inputs but checks for female average, should return 0. Decided to test this as an extreme case and an easier way to make sure function only calculates average of given gender.

- testAverage4 - test when input is incorrect, should return the statement ”in g enter male or female”. Decided to test it to see what happens when there is human error.
- testAllocate1 - test when all files are empty, should return a dictionary of departments with empty list. Decided to test with same reasoning as testSort1.
- testAllocate2 - test with normal inputs to make sure implementation is correct
- testAllocate3 - test when no one has free choice, to make sure if there is no free choice everything still runs smoothly.
- testAllocate4 - test when some people get third choice stream. Decided to test this to further insure my program works correctly, in a more difficult normal input.
- testAllocate5 - test when every student has gpa below 4.0 or gpa above 12.0, should return dictinary of departments with empty list. Decided to test to make sure students that are allocated have the correct gpa to enter second year.

2 Results of Testing Partner’s Code

My partner’s code had variables that contain his/her textfiles so when I tested it, it said it could not identify variable, so I replaced the variable with L for the average function.

- testSort1 - passed
- testSort2 - passed
- testAverage1 - failed, because I had a condition when there are no students, returns zero but partner did not have it, so it gave a zerodivision error.
- testAverage2 - passed
- testAverage3 - failed, same reason as testAverage1
- testAverage4 - failed, had condition for when user types wrong input, while partner did not
- testAllocate1 - passed
- The rest of the testAllocate failed, mainly due to the reason I assumed people who have 4.0 gpa can enter second year, while my partner assumed they are ineligible to enter second year.

3 Discussion of Test Results

What I learned by doing this exercise is how to improve my testing abilities by having a mindset of how can I destroy this program. Also it showed me to not test for inputs that are expected but also for the unexpected inputs because not every input is going to be correct.

Assumptions I Made:

- when all 3 student's choices are full decided not to allocate student, since if that was the scenario in real life a student would not be allocated to a random department.
- Assumed that people who have 4.0 gpa are eligible to enter second year
- Assumed that students in textfile had all correct information
- Assumed for the most part there was no human error when entering inputs
- Assumed people who have gpa higher than 12 are not allocated to second year

3.1 Problems with Original Code

In terms of my testing, my program passed all the test cases but it is not perfect since it does not account for majority of input errors, such as inputing a something besides a list, inputting incorrect information etc...

3.2 Problems with Partner's Code

For my partner's program first problem was that he/she had a variable that conatains his/her textfiles which made me have to replace the variable with L for the average function. Also my partner had different assumptions then I did leading to some test cases failing such as assumming 4.0 gpa is ineligible.

4 Critique of Design Specification

- Liked how we can choose our own data structure for textfile, makes it easier to implement it
- Didn't enjoy the assumptions we had to make in certain scenarios such as deciding what to do if top 3 choices were full
- Wish I knew what types of input errors were required to be fixed since there are a lot of possible errors, so wish I knew the types of errors needed to be fixed

5 Answers to Questions

(a) To make average function more general, one idea is to have the option to calculate the average of all students as well. Also, another idea is to have option to calculate median and mode to get more sets of data that can be useful for the university. To make sort more general instead of sorting just by gpa, you can sort by alphabetical order.

(b) Aliasing means that you use assign the dictionary to a variable and then assign that variable to another variable. This might be a problem for dictionaries because if you change the content of the second variable it changes the content of the first variable. To guard against this problem is to use tuples since they immutable meaning you can't change the content of the students.

(c) 2 test cases for read allocation can be first normal inputs to test if ReadAllocationData works at all and a second test case can be maybe a student info is half filled to check what happens during the extreme cases and when everything is not perfect. I believe CalcModule was selected over ReadAllocationData because when you test CalcModule, it would be easier for to generate the lists from ReadAllocationData, then to actually hardcode lists for test cases, meaning it would be easier to test CalcModule by making sure ReadAllocationData is correct.

(d) The problem with using strings in these cases, is that the user has to input the exact same string that you used, meaning if for example you set male with a lower case and the user typed male with an uppercase, the function would not work, this also applies to keys in the dictionary. A better way to possibly do it is make the input all lower case in the function in order to avoid cases where a person inputs "Male" instead of "male".

(e) Other options could be creating a class, in which each student can be represented as an object with all their requirements used as variables in the class. I believe changing the data structure would be better, to possibly for example because you can represent an student as an actual object, instead of a string/key.

(f) You wouldn't have to change the code for calcmodule since you don't change anything in the list of choices, but for read allocation data you would have to change the code, since tuples are immutable so you wouldn't be able to append the choices to the tuple. So for the question with custom class, again it's the same reasoning as the sentence above, because you don't modify the data of choices so changing the code is not necessary, but in read allocation data, you have to modify the data, meaning you would have to change the code. Also if you change other list like the list for departments, then you would have to change the code of calcmodule since you are constantly modifying the list.

F Code for ReadAllocationData.py

```
## @file ReadAllocationData.py
#  @author Bill Nguyen
#  @brief Reads Data to generate student information entering second year
#  @date January 18, 2019



## @brief Gets all the students in textfile and makes a list filled with dictionaries representing
    students
#  @param s is textfile representing students entering second year
#  @return List with dictionaries inside, representing students entering second year
def readStdnts(s):
    file = open(s, "r")
    values = ["macid", "fname", "lname", "gender", "gpa", "choices"]
    student_list = []
    file_list = file.readlines()
    for line in file_list:
        split_list = line.split(",")
        student_dict = {}
        for i in range(0, 6):
            if i == 4:
                gpa = float(split_list[i])
                student_dict[values[i]] = gpa
            elif i == 5:
                choices = split_list[i].split()
                student_dict[values[i]] = choices
            else:
                student_dict[values[i]] = split_list[i]
        student_list.append(student_dict)
    file.close()
    return student_list

## @brief Puts all students with free choice into a list
#  @param s is textfile containing students who have free choice
#  @return List with all students who have free choice
def readFreeChoice(s):
    file = open(s, "r")
    free_list = []
    for line in file:
        free_list.append(line.strip("\n"))
    file.close()
    return free_list
## @brief Creates a dictionary that contains the capacity of each department
#  @param s is a textfile that contains all department capacities
#  @return Dictionary of all department capacities
def readDeptCapacity(s):
    file = open(s, "r")
    dept_dict = {}
    file_list = file.readlines()
    for dept in file_list:
        split_list = dept.split(",")
        dept_dict[split_list[0]] = int(split_list[1].strip("\n"))
    file.close()
    return dept_dict
```

G Code for CalcModule.py

```
## @file CalcModule.py
#  @author Bill Nguyen
#  @brief Sorts, calculates averages and allocates students entering second year
#  @date January 18, 2019

from ReadAllocationData import *

## @brief Sorts list of students in descending order
#  @param S is list of dictionaries generated from the readStdnts function
#  @return List of dictionaries that is sorted in descending order based on gpa
def sort(S):
    for j in range(len(S)):
        index = 0
        while index < (len(S) - 1):
            if (S[index]["gpa"] > S[index+1]["gpa"]) or (S[index]["gpa"] == S[index+1]["gpa"]):
                index += 1
            else:
                temp = S[index]
                S[index] = S[index + 1]
                S[index+1] = temp
    return S

## @brief Calculates average of all males or females students
#  @param L is a list of dictionaries generated from readStdnts function
#  @param g is a string that has the input male or female
#  @return Average of all males or females students

def average(L, g):
    sum_g = 0
    count = 0
    if g != "male" and g != "female":
        return "in g enter male or female"
    for student in L:
        if student["gender"] == g:
            sum_g += student["gpa"]
            count += 1
    if count == 0:
        return 0
    else:
        return sum_g/count

## @brief Allocates all students to second year stream
#  @param S a list of dictionaries generated by readStdnts function
#  @param F a list that contains students with free choice
#  @param C a dictionary that contains all department capacities
#  @return  a dictionary that allocates all qualified students to their second year stream
def allocate(S, F, C):
    choice_dict = { "civil": [], "chemical": [],"electrical": [],"software": [],"materials":
        [],"mechanical": [],"engphys": [] }
    freeChoice_list = []
    S_list = []
    for student in S:
        if (student["gpa"] >= 4.0) and (student["macid"] in F) and (student["gpa"] <= 12.0):
            freeChoice_list.append(student)
        elif (student["gpa"] >= 4.0) and (student["gpa"] <= 12.0):
            S_list.append(student)

    freeChoice_list = sort(freeChoice_list)

    for student in freeChoice_list:
        for x in student["choices"]:
            if C[x] > len(choice_dict[x]):
                choice_dict[x].append(student)
                break

    sorted_list = sort(S_list)

    for student in sorted_list:
        for x in student["choices"]:
            if C[x] > len(choice_dict[x]):
                choice_dict[x].append(student)
                break

    return choice_dict
```

H Code for testCalc.py

```
## @file testCalc.py
#  @author Bill Nguyen
#  @brief Testing CalcModule.py functions
#  @date January 18,2019

from ReadAllocationData import *
from CalcModule import *

#Referenced Assignment 1 2018 Solution For Format of testCalc.py

def assertionEqual(testcase, result, name):
    if testcase == result:
        print("%s passed" % (name))
    else:
        print("%s failed, correct result is: %s" % (name, result))



def testSort1():
    sort_list = sort(readStdnts("src/TextFiles/ReadStdntsTest1.txt"))
    #TestSort1: when ReadStdnts is empty
    #Results suppose to be []
    assertionEqual(sort_list, [], "TestSort1")

def testSort2():
    #TestSort2: normal input
    sort_list = sort(readStdnts("src/TextFiles/ReadStdntsTest2.txt"))
    correct = [{'macid': 'macid3',
                'fname': 'firt3',
                'lname': 'last3',
                'gender': 'female',
                'gpa': 12.0,
                'choices': ['software', 'mechanical', 'electrical']},
               {'macid': 'macid2',
                'fname': 'first2',
                'lname': 'last2',
                'gender': 'male',
                'gpa': 11.0,
                'choices': ['civil', 'chemical', 'engphys']},
               {'macid': 'macid4',
                'fname': 'firt4',
                'lname': 'last4',
                'gender': 'male',
                'gpa': 11.0,
                'choices': ['chemical', 'mechanical', 'electrical']},
               {'macid': 'macid1',
                'fname': 'first1',
                'lname': 'last1',
                'gender': 'male',
                'gpa': 10.8,
                'choices': ['software', 'mechanical', 'electrical']},
               {'macid': 'macid5',
                'fname': 'firt5',
                'lname': 'last5',
                'gender': 'male',
                'gpa': 9.0,
                'choices': ['mechanical', 'electrical', 'materials']},
               {'macid': 'macid6',
                'fname': 'firt6',
                'lname': 'last6',
                'gender': 'female',
                'gpa': 4.0,
                'choices': ['engphys', 'mechanical', 'materials']}]
    assertionEqual(sort_list, correct, "TestSort2")

def testAverage1():
    avg = average(readStdnts("src/TextFiles/ReadStdntsTest1.txt"), "male")
    #testing when text file is empty
    assertionEqual(avg, 0, "TestAverage1")

def testAverage2():
    avg = average(readStdnts("src/TextFiles/ReadStdntsTest2.txt"), "male")
    correct = 10.45
    #testing with normal inputs
    assertionEqual(avg, correct, "TestAverage2")
```

```python
def testAverage3():
    avg = average(readStdnts("src/TextFiles/ReadStdntsTest3.txt"), "female")
    #testing with only male inputs but checking for female average
    assertionEqual(avg, 0, "TestAverage3")
def testAverage4():
    avg = average(readStdnts("src/TextFiles/ReadStdntsTest2.txt"), "hello")
    #testing when user does not input male or female in g
    assertionEqual(avg, "in g enter male or female", "TestAverage4")

def testAllocate1():
    allo = allocate(readStdnts("src/TextFiles/ReadStdntsTest1.txt"),
        readFreeChoice("src/TextFiles/ReadFreeChoiceTest1.txt"),
        readDeptCapacity("src/TextFiles/ReadDeptCapacityTest1.txt"))
    #testing when every textfiles are empty
    correct = {'civil': [],
        'chemical': [],
        'electrical': [],
        'software': [],
        'materials': [],
        'mechanical': [],
        'engphys': []}
    assertionEqual(allo, correct, "TestAllocate1")

def testAllocate2():
    allo = allocate(readStdnts("src/TextFiles/ReadStdntsTest2.txt"),
        readFreeChoice("src/TextFiles/ReadFreeChoiceTest2.txt"),
        readDeptCapacity("src/TextFiles/ReadDeptCapacityTest2.txt"))
    #testing with normal inputs
    correct = {'civil': [{'macid': 'macid2', 'fname': 'first2', 'lname': 'last2', 'gender': 'male',
        'gpa': 11.0, 'choices': ['civil', 'chemical', 'engphys']}], 'chemical': [{'macid': 'macid4',
        'fname': 'firt4', 'lname': 'last4', 'gender': 'male', 'gpa': 11.0, 'choices': ['chemical',
        'mechanical', 'electrical']}], 'electrical': [], 'software': [{'macid': 'macid1', 'fname':
        'first1', 'lname': 'last1', 'gender': 'male', 'gpa': 10.8, 'choices': ['software',
        'mechanical', 'electrical']}], 'materials': [], 'mechanical': [{'macid': 'macid3', 'fname':
        'firt3', 'lname': 'last3', 'gender': 'female', 'gpa': 12.0, 'choices': ['software',
        'mechanical', 'electrical']}, {'macid': 'macid5', 'fname': 'firt5', 'lname': 'last5',
        'gender': 'male', 'gpa': 9.0, 'choices': ['mechanical', 'electrical', 'materials']}],
        'engphys': [{'macid': 'macid6', 'fname': 'firt6', 'lname': 'last6', 'gender': 'female',
        'gpa': 4.0, 'choices': ['engphys', 'mechanical', 'materials']}]}
    assertionEqual(allo, correct, "TestAllocate2")

def testAllocate3():
    allo = allocate(readStdnts("src/TextFiles/ReadStdntsTest2.txt"),
        readFreeChoice("src/TextFiles/ReadFreeChoiceTest1.txt"),
        readDeptCapacity("src/TextFiles/ReadDeptCapacityTest2.txt"))
    #testing when ReadFreeChoice text file is empty
    correct = {'civil': [{'macid': 'macid2', 'fname': 'first2', 'lname': 'last2', 'gender': 'male',
        'gpa': 11.0, 'choices': ['civil', 'chemical', 'engphys']}], 'chemical': [{'macid': 'macid4',
        'fname': 'firt4', 'lname': 'last4', 'gender': 'male', 'gpa': 11.0, 'choices': ['chemical',
        'mechanical', 'electrical']}], 'electrical': [], 'software': [{'macid': 'macid3', 'fname':
        'firt3', 'lname': 'last3', 'gender': 'female', 'gpa': 12.0, 'choices': ['software',
        'mechanical', 'electrical']}], 'materials': [], 'mechanical': [{'macid': 'macid1', 'fname':
        'first1', 'lname': 'last1', 'gender': 'male', 'gpa': 10.8, 'choices': ['software',
        'mechanical', 'electrical']}, {'macid': 'macid5', 'fname': 'firt5', 'lname': 'last5',
        'gender': 'male', 'gpa': 9.0, 'choices': ['mechanical', 'electrical', 'materials']}],
        'engphys': [{'macid': 'macid6', 'fname': 'firt6', 'lname': 'last6', 'gender': 'female',
        'gpa': 4.0, 'choices': ['engphys', 'mechanical', 'materials']}]}
    assertionEqual(allo, correct, "TestAllocate3")

def testAllocate4():
    allo = allocate(readStdnts("src/TextFiles/ReadStdntsTest4.txt"),
        readFreeChoice("src/TextFiles/ReadFreeChoiceTest2.txt"),
        readDeptCapacity("src/TextFiles/ReadDeptCapacityTest2.txt"))
    #testing when some people go their third choice stream
    correct = {'civil': [], 'chemical': [{'macid': 'macid2', 'fname': 'first2', 'lname': 'last2',
        'gender': 'male', 'gpa': 11.0, 'choices': ['software', 'chemical', 'engphys']}],
        'electrical': [{'macid': 'macid5', 'fname': 'firt5', 'lname': 'last5', 'gender': 'male',
        'gpa': 9.0, 'choices': ['mechanical', 'electrical', 'materials']}], 'software': [{'macid':
        'macid1', 'fname': 'first1', 'lname': 'last1', 'gender': 'male', 'gpa': 10.8, 'choices':
        ['software', 'mechanical', 'electrical']}], 'materials': [{'macid': 'macid6', 'fname':
        'firt6', 'lname': 'last6', 'gender': 'female', 'gpa': 4.0, 'choices': ['mechanical',
        'software', 'materials']}], 'mechanical': [{'macid': 'macid4', 'fname': 'firt4', 'lname':
        'last4', 'gender': 'male', 'gpa': 11.0, 'choices': ['mechanical', 'materials',
        'electrical']}, {'macid': 'macid3', 'fname': 'firt3', 'lname': 'last3', 'gender': 'female',
        'gpa': 12.0, 'choices': ['software', 'mechanical', 'electrical']}], 'engphys': []}
    assertionEqual(allo, correct, "TestAllocate4")

def testAllocate5():
    allo = allocate(readStdnts("src/TextFiles/ReadStdntsTest5.txt"),
        readFreeChoice("src/TextFiles/ReadFreeChoiceTest2.txt"),
        readDeptCapacity("src/TextFiles/ReadDeptCapacityTest2.txt"))
    #testing when every student has a gpa below 4.0 or gpa above 12.0
    correct = {'civil': [],
        'chemical': [],
        'electrical': [],
        'software': [],
        'materials': [],
        'mechanical': [],
        'engphys': []}
    assertionEqual(allo, correct, "TestAllocate5")

def test():
    testSort1()
    testSort2()
    testAverage1()
    testAverage2()
    testAverage3()
    testAverage4()
    testAllocate1()
    testAllocate2()
    testAllocate3()
    testAllocate4()
    testAllocate5()

test()
```

I Code for Partner's CalcModule.py

```
## @file CalcModule.py
#  @author Kevin Zhou
#  @brief completes various calculations on the list of students created
#         using the functions in the previous module.
#  @date 16/01/2019

import ReadAllocationData
lib1 = ReadAllocationData.readStdnts('library1.txt')
lib2 = ReadAllocationData.readFreeChoice('library3.txt')
lib3 = ReadAllocationData.readDeptCapacity('library2.txt')
## @brief A function that sorts the students in decending order of GPA.
#  @details It is just a general bubble sort algorithm and it takes
#            the GPA of all students and place them in decending order
#            into the list.


def sort(S):
    for num in range(len(S)-1,0,-1):
        for i in range(num):
            if S[i]['gpa']<S[i+1]['gpa']:
                temp = S[i]
                S[i] = S[i+1]
                S[i+1] = temp
    # print(S)
    return(S)
# sort(lib1)

## @brief A function that calculates the average of the students
#    based on input gender.
#  @details It takes a list and a string, which is the gender of the students,
#            and it loops through the student data to grab the matched gender.
#            Then, it just adds the gpa of those students and divides by
#            the number of the students who matches the gender.

def average(L, g):
    sum_up = 0
    gender = []
    for i in lib1:
        if i['gender'] == g:
            gender.append(i)
    for j in gender:
        sum_up = sum_up + j['gpa']
    avg = sum_up / len(gender)
    # print(avg)
    return(avg)

# average(lib1, 'Female')

## @brief A function that allocate the students in the library
#    into their choices based on various things.
#  @details It loops through the data and first put the students
#           with free choice into a list, and for the rest of those
#           students, they will be put into another list. For each list,
#           first it checks if the gpa is greater than 4, then it starts
#           placing students into their choice. For each operation, it will
#           decrease the department capacity. The 'passed' list will be sorted
#           to match the requirement.
def allocate(S,F,C):
    out = {'civil':[], 'chemical':[], 'electrical':[], 'mechanical':[], 'software':[], 'materials':[],
        'engphys':[]}
    passed = []
    free = []

    for student in S:
        if student['macid'] in F:
            free.append(student)
        else:
            passed.append(student)

    for i in range(len(free)):
        if free[i]['gpa'] > 4.0:

            if C[(free[i]['choices'][0])] > 0:
                out[(free[i]['choices'][0])].append(free[i])
                C[(free[i]['choices'][0])] -= 1
            elif C[(free[i]['choices'][1])] > 0:
                out[(free[i]['choices'][1])].append(free[i])
                C[(free[i]['choices'][1])] -= 1

            elif C[(free[i]['choices'][2])] > 0:
                out[(free[i]['choices'][2])].append(free[i])
                C[(free[i]['choices'][2])] -= 1


    passed = sort(passed)
    for j in range(len(passed)):
        if passed[j]['gpa'] > 4.0:

            if C[(passed[j]['choices'][0])] > 0:
                out[(passed[j]['choices'][0])].append(passed[j])
                C[(passed[j]['choices'][0])] -= 1

            elif C[(passed[j]['choices'][1])] > 0:
                out[(passed[j]['choices'][1])].append(passed[j])
                C[(passed[j]['choices'][1])] -= 1

            elif C[(passed[j]['choices'][2])] > 0:
                out[(passed[j]['choices'][2])].append(passed[j])
                C[(passed[j]['choices'][2])] -= 1

    # print(out)
    return out

# allocate(lib1, lib2, lib3)
```

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
```