Assignment 1 Solution

Alice Ip and ipa1

January 25, 2019

Part 1 of this software design exercise was to write modules to allocate engineering students from first year into their second year programs. One module reads relevant data from files and the other performs operations on the data. In addition there was a python file for the test driver. In this report, I will summarize and discuss the results of the implementation and testing of my modules as well as a comparison of the module of another student in the course.

1 Testing of the Original Program

To design test cases for each function, I first created a test case for the simplest form, such as having an empty list of dictionaries of students and then I created test cases to test specifications from the assignment, and calculations from the functions. All test cases passed. Assumptions made include:

- Only students with a gpa higher than 4 will be allocated, students with a gpa of 4 or less will not be allocated
- All free choice students will be granted their first choice regardless of capacity
- Non-free choice students who were not able to get into any of their top three choices would be randomly allocated into a department
- There is enough capacity to fit all students in at least one program

2 Results of Testing Partner's Code

After a successful compile of my "testCalc.py" test module, the partner's "CalcModule.py" module passed 5/9 tests. Assumptions that the partner made include:

- Only students with a gpa higher than 4 will be allocated, students with a gpa of 4 or less will not be allocated
- If the capacity of a department that a free choice student chose is full, they will be allocated according to their remaining two choices
- If the capacities of all the departments that a student chose is full, they will be put into a "deptFull" list that holds all the unallocated students

Since my partner made the same assumption regarding the GPA, some of the expected results from the test cases were correct. However, my partner assumed that capacity was a higher priority than free chocie, and did not allocate free choice students to their first choice if the capacity was full. As a result, failed the test that tested for free choice allocation to first choice regardless of capacity. In the case that any of the inputs were empty, my partner would return a string "Data is empty" while I assumed that a empty dictionary with no allocations was sufficient, so my partner failed this test. My partner performed integer division when calculating the average, and as a result, the output did not fit the approximation error range that I had implemented in my test cases for average, failing this test. I realised while testing for regular students being allocated to second choice due to capacity, that a separate test case with a separate input should have been made, as the errors from other assumptions affected this test case, resulting in another failed test.

3 Discussion of Test Results

3.1 Problems with Original Code

After testing the partner code, I discovered that I had read the requirements specifications wrong, and put the wrong type for "gpa" and "capacity". As a result, although all the tests pass for my module, they caused errors in the partner "CalcModule" files similar to the image below. Also, I was able to compare and analyze my approach to the specifications. Rather than forcing students into a program with full capacity or into a program that is not in their top three choices, it may have been better to not allocate the students, and throw an exception. This method would raise awareness to the problem the school has with their capacities and may increase the satisfaction of the students.

3.2 Problems with Partner's Code

A common theme I noticed in my partner's "CalcModule.py" file is the excessive use of lists to accomplish a task. In the average function, my partner created two lists, one to hold all the students of the gender being calculated, and another to hold the GPA values of the students of the correct gender. For the purposes of this function, I think that it is sufficient to iterate through the list once, check the gender, add the GPA value into a variable that will hold the total, while keeping track of the number of students that are the correct gender. To calculate the average value, my partner used integer division, resulting in a loss of precision. It is not stated how the result of this function will be used, however in a typical situation where a average is used, usually, precision may be important for drawing conclusions.

4 Critique of Design Specification

The design specification through natural language was ambiguous and resulted in a substantial amount of assumptions. Depending on the approach that the programmer designs, the results of the program may be far from the expectations of assigner. For example, it is stated "The algorithm for the allocation will allocate all students with a gpa greater than 4.0. Those with less than 4.0 will not be allocated." However, these two specifications do not indicate the correct course of action for a student with a GPA of exactly 4.0. Another example would be the lack of specification for situations where capacity is full for free students as well as for regular students. The assignee lacks the information needed to decide how such situations should be handled. Although the specifications for the input and output types of each function was very clear, to ensure reliability and correctness of the program, more information about the expectations of the output should be included.

5 Answers to Questions

(a) How could you make function average(L,g) more general? That is, can you specify a similar function, but one that is more versatile/flexible than the given function? The new function should be capable of the identical behaviour as average(L, g) but also have other capabilities. Along a similar line of thinking, how could you make the sort(S) more general?

To make average(L,g) more general, To make sort(S) more general, a new parameter could be taken to determine whether the sort is in ascending or descending order. sort(S,order)

(b) The assignment states that you can assume that aliasing will not occur with the dictionaries in your lists. What does aliasing mean in this context? In general, could aliasing be a concern with dicts? How might you guard against potential problems?

Aliasing in this context, may happen whenever two variables refer to the same dictionary object, and a change is made to a value using one variable that results in a change in value of the other variable. In general, this may be an issue if someone wanted to make a copy of a dictionary, using "=" operator or if someone tries to assign a dictionary value to another dictionary or list. To guard against potential problems, good knowledge of which data types are mutable, good testing of functions, and careful usage of assignment variables when using mutable data types can help.

(c) The assignment did not require you to test the ReadAllocationData.py module. Describe some potential test cases you could have used to build confidence in this module. Of the two modules, why do you think CalcModule.py was selected over ReadAllocationData.py as the one you should test?

Potential test cases to build confidence in the "ReadAllocationData.py" module could include testing to ensure that for each output, the correct types are present. For example, possible testing to make sure that the GPA is a float value, the first name is a string value, and that capacity is a positive integer. Other test cases could include testing to make sure the values in the textfiles are of the correct format. For example, whether the macid started with alphabet letters and ended with a number if there is a number.The Implementation of the functions in "ReadAllocationData.py" was not significant as long as it produced output in the specified format. As well, there was little to no manipulation of the data that was processed in these functions, and ambiguities would be insignificant. "CalcModule'.py" on the other hand used a combination of many different data types such as dictionaries and lists, and had much more constraints and requirements to meet. Consequently, bugs, errors, and unexpected situations were more likely to exist and cause an undesired outcome as a result of function use.

(d) The assignment uses strings as the keys in several dictionaries. It also uses strings to represent members of finite sets. For instance, the strings "male" and "female" are used to represent elements of the set male, female'. What are the problems with using strings in this way? What would be a better approach?

A problem with using strings in this way, is that it must be referred to, and used as a string. Using strings as dictionary keys cause typing to be more ambiguous. A better approach would be to use a enumerated type, where "male" and "female" are the values of a user defined data type "gender".

(e) A dictionary isn't the only option to implement records and structs in Python, where records and structs correspond to the mathematical notion of a tuple. What are other options for implementing the mathematical notion of tuples in Python? Would you recommend changing the data structure used in the code modules? How would you change it?

Other ways of implementing records and structs include writing a custom class. By using a class, you can ensure that each object provides the same set of fields of the correct type. Fields stored on classes are mutable, and new fields can be added freely. Properties of a class can be accessed easily. Another method would be to use the types.SimpleNamespaceClass. This method allows you to access, modify, add, and delete attributes easily. I would recommend changing the data structure used in the code modules to the typing.NamedTupleClass, for this particular problem, since the modules use simple types that are already in Python. This Class allows you to define the types of values in the objects, removing a large portion of possible ambiguity, while keeping the structure of each object simple. The fields are immutable, preventing possible change of values, which is not required for the modules in the assignment.

(f) In the specification the student's preferred choices are listed as 'choices': [string, string, string]. If the list of strings was changed to a different data structure like a tuple, would the CalcModule.py module need to be modified? Consider instead if a custom class (Abstract Data Type (ADT)) was written for students, and the CalcModule.py module was modified accordingly. This custom class provides a method that returns the next choice and another method that returns True when there are no more choices. In this new case, if the data structure inside the custom class changes, say from lists to tuples, will the CalcModule.py module need to be modified? Please justify your answer

Tuples and lists are accessed in the same way, through indexing. However, if a dictionary is used, the CalcModule.py may need to be modified to access the correct values. If a custom class was written for students with the stated functions, if the data structure inside the custom class changes, the CalcModule.py module would not need to be modified, since the CalcModule

F Code for ReadAllocationData.py

```
## @file ReadAllocationData.py
#  @author Alice Ip
#  @brief Reads in files and organizes the information into a specific format
#  @date 2019−01−13

## @brief Formats a file input into a list of dictionaries
#  @param s string corresponding to a filename
#  @return list of dictionaries
def readStdnts(s):
        iFile = open(s, "r")
        allRecords = []
        sRecord = {}

        for line in iFile:
                sLine = line.strip()
                sLine = sLine.split(" ")

                sLineLen= len(sLine)
                while (sLineLen < 8):
                        sLine.append("")
                        sLineLen+=1


                sRecord = dict(macid=sLine[0], fname=sLine[1], lname=sLine[2], gender=sLine[3],
                    gpa=float(sLine[4]), choices=[sLine[5],sLine[6],sLine[7]])
                allRecords.append(sRecord);
        return allRecords

## @brief Formats a file input into a list of macids of students who get free choice
#  @param s string corresponding to a filename
#  @return list of macIds
def readFreeChoice(s):
        freeList=[]
        iFile = open(s, "r")
        for line in iFile:
                sLine = line.strip()
                sLine = sLine.split(" ")
                freeList.append(sLine[0])
        return freeList

## @brief Formats a file input into a dictionary of departments and capacities
#  @param s string corresponding to a filename
#  @return dictionary of departments and capacities
def readDeptCapacity(s):
        deptList={}
        iFile = open(s, "r")

        for line in iFile:
                sLine = line.strip()
                sLine = sLine.split(" ")
                deptList[sLine[0]] = int(sLine[1])
        return deptList
```

G Code for CalcModule.py

```
## @file CalcModule.py
#  @author Alice Ip
#  @brief Completes various calculations on the list of students created using functions in
     ReadAllocationDaya.py
#  @date 2019-01-13

## @brief Sorts a list of student dictionaries based on gpa
#  @details Using the gpa key of the student dictionaries, the positions of students in the list will
     be rearranged to be in order of gpa
#  @param S a list of the dictionaries of students
#  @return a list of the dictionaries of students
def sort(s):
        list_length = len(s)
        for i in range(list_length -1,0,-1):
                for j in range(0,i):
                        if ((s[j])['gpa']) < ((s[j+1])['gpa']):
                                temp = s[j]
                                s[j] = s[j+1]
                                s[j+1] = temp
        return s

## @brief Calculates the average gpa of a list of students
#  @details Given a list of students and their gender, the average is calculated
#  @param L a list of dictionaries created by function readStdnts(s)
#  @param g a string representing male or female
#  @return average gpa of a list of students
def average(L, g):
        stud_average = 0
        person_count = 0
        for student in L:
                if student['gender'] == g:
                        stud_average  += (student['gpa'])
                        person_count +=1
        if person_count ==0:
                return 0
        else:
                return (stud_average/person_count)

## @brief Allocates students into departments
#  @details Given a list of dictionaries of students, list of students with free choice, and capacity,
     returns dictionary of department capacities
#  @param S a list of the dictionaries of students
#  @param F a list of students with free choice
#  @param C a dictionary of department capacities
#  @return a dictionary with departments as keys and students in the program in a list as the value

#  Assumption: Only students with a gpa higher than 4 will be allocated, students with a gpa of 4 or
     less will not be allocated
#  Assumption: All free choice students will be granted their first choice regardless of capacity

def allocate(S,F,C):
        allocated={'civil':[], 'chemical':[], 'electrical':[], 'mechanical':[], 'software':[],
             'materials':[], 'engphys':[]}
        student_allocated = False

        allocating = S.copy()
        allocating = sort(allocating)

        #Allocate the students with free choice by finding their record in the general list, checking
             gpa, and capacity of desired choice
        for fc_student in F: #Iterate through all students with free choice
                for all_student in allocating: #Iterate through all students in general list
                        if (all_student)['macid'] == fc_student: #If macid matches
                                if (all_student['gpa']) > 4: #Check if their gpa entry is greater than
                                     4
                                        student_choice = ((all_student)['choices'])[0]
                                        allocated[student_choice].append((all_student)['macid']) #Add
                                             student to department
                                        allocating.remove(all_student) #Remove student from general
                                             list to keep track of who has been allocated already

                                else: #Student did not have at least a 4 gpa
                                        allocating.remove(all_student)
```

```
#Allocate the remaining students, the list is already sorted and in order of gpa
for rem_student in allocating:
        if  (rem_student)['gpa'] >= 4:
                for choices in (rem_student)['choices']:
                        for all_dept in C:
                                if all_dept == choices:
                                        if len((allocated[all_dept])) < (C[all_dept]):
                                                #print(all_dept, "is an eligible choice for",
                                                     (rem_student)['macid'])
                                                allocated[all_dept].append((rem_student)['macid'])
                                                student_allocated = True
                                                allocating.remove(rem_student)
                                                break
                        if student_allocated == True:
                                student_allocated = False
                                break
return(allocated)
```

H Code for testCalc.py

```
## @file testCalc.py
#  @author Alice Ip
#  @brief Test file to test functions in CalcModule
#  @date 2019-01-18

from ReadAllocationData import *
from CalcModule import *

ave_test_1= [{'macid': 'ipa1', 'fname': 'Alice', 'lname': 'Ip', 'gender': 'female', 'gpa': 7,
     'choices': ['materials', 'engphys', 'civil']},
{'macid': 'ipk2', 'fname': 'Kevin', 'lname': 'Ip', 'gender': 'male', 'gpa': 8, 'choices':
     ['materials', 'software', '']},
{'macid': 'ipe3', 'fname': 'Eric', 'lname': 'Ip', 'gender': 'male', 'gpa': 12, 'choices': ['civil',
     'chemical', 'electrical']},
{'macid': 'ipa3', 'fname': 'Alex', 'lname': 'Ip', 'gender': 'female', 'gpa': 5, 'choices':
     ['materials', 'chemical', 'software']}]

ave_test_2= [{'macid': 'ipa1', 'fname': 'Alice', 'lname': 'Ip', 'gender': 'male', 'gpa': 7, 'choices':
     ['materials', 'engphys', 'civil']},
{'macid': 'ipk2', 'fname': 'Kevin', 'lname': 'Ip', 'gender': 'male', 'gpa': 8, 'choices':
     ['materials', 'software', '']},
{'macid': 'ipe3', 'fname': 'Eric', 'lname': 'Ip', 'gender': 'male', 'gpa': 12, 'choices': ['civil',
     'chemical', 'electrical']},
{'macid': 'ipa3', 'fname': 'Alex', 'lname': 'Ip', 'gender': 'male', 'gpa': 5, 'choices': ['materials',
     'chemical', 'software']}]

ave_test_3= [{'macid': 'ipa1', 'fname': 'Alice', 'lname': 'Ip', 'gender': 'female', 'gpa': 7,
     'choices': ['materials', 'engphys', 'civil']},
{'macid': 'ipk2', 'fname': 'Kevin', 'lname': 'Ip', 'gender': 'male', 'gpa': 8, 'choices':
     ['materials', 'software', '']},
{'macid': 'ipe3', 'fname': 'Eric', 'lname': 'Ip', 'gender': 'male', 'gpa': 12, 'choices': ['civil',
     'chemical', 'electrical']},
{'macid': 'ipa3', 'fname': 'Alex', 'lname': 'Ip', 'gender': 'male', 'gpa': 5, 'choices': ['materials',
     'chemical', 'software']}]

sort_test_1= [{'macid': 'ipa1', 'fname': 'Alice', 'lname': 'Ip', 'gender': 'female', 'gpa': 7,
     'choices': ['materials', 'engphys', 'civil']},
 {'macid': 'ipk2', 'fname': 'Kevin', 'lname': 'Ip', 'gender': 'male', 'gpa': 8, 'choices':
      ['materials', 'software', '']},
  {'macid': 'ipe3', 'fname': 'Eric', 'lname': 'Ip', 'gender': 'male', 'gpa': 12, 'choices': ['civil',
       'chemical', 'electrical']}]

sort_test_2= [{'macid': 'ipa1', 'fname': 'Alice', 'lname': 'Ip', 'gender': 'female', 'gpa': 7,
     'choices': ['materials', 'engphys', 'civil']},
 {'macid': 'ipk2', 'fname': 'Kevin', 'lname': 'Ip', 'gender': 'male', 'gpa': 8, 'choices':
      ['materials', 'software', '']},
  {'macid': 'ipe3', 'fname': 'Eric', 'lname': 'Ip', 'gender': 'male', 'gpa': 7, 'choices': ['civil',
       'chemical', 'electrical']}]

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

def assertionApproximatelyEqual(fname, test, result, error, name):
  if abs(test - result) < error:
    print("Test passed, %s : Actual: %s  Approximate: %s, %s " % (fname, test, result, name))
  else:
    print("Test failed, %s : Actual: %s  Approximate: %s, %s " % (fname, test, result, name))


def test_sort_1():
        test_list = sort(sort_test_1)
        list_sorted = True
        for student in range(len(test_list)-1,0,-1):
                if int(test_list[student]['gpa']) > int(test_list[student-1]['gpa']):
                        list_sorted = False
                        break
        assertionEqual("sort(s)", True, list_sorted, "sorted regular list")

def test_sort_2():
        test_list = sort(sort_test_2)
        list_sorted = True
        for student in range(len(test_list)-1,0,-1):
                if int(test_list[student]['gpa']) > int(test_list[student-1]['gpa']):
                        list_sorted = False
                        break
        assertionEqual("sort(s)", True, list_sorted , "sorted list with two duplicate gpa values")

def test_average_1():
        test_list = ave_test_1
        assertionApproximatelyEqual("average(L, g)", 6,average(test_list ,"female"), 0.000005,
            "integer")

def test_average_2():
        test_list = ave_test_2
        assertionApproximatelyEqual("average(L, g)", 0,average(test_list ,"female"), 0.000005, "no
            values in list matching gender")

def test_average_3():
        test_list = ave_test_3
        assertionApproximatelyEqual("average(L, g)", 8.3333333,average(test_list ,"male"), 0.000005,
            "floating point average test")

def test_allocate_1():
        test_file_1 = "alllist.txt"
        test_file_2 = "fclist.txt"
        test_file_3 = "deptlist.txt"

        S_test = readStdnts(test_file_1)
        F_test = readFreeChoice(test_file_2)
        C_test = readDeptCapacity(test_file_3)

        result = allocate(S_test ,F_test ,C_test)
        assertionEqual("allocate(S,F,C)", ['ipa1', 'ipk2'], result["materials"] , "free choice
            allocated to first choice regardless of capacity")
        assertionEqual("allocate(S,F,C)", ['ipe3'], result["chemical"] , "regular student allocated to
            second choice due to capacity")
        assertionEqual("allocate(S,F,C)", [], result["electrical"] , "free choice and regular students
            must have a gpa of 4 to be allocated")

def test_allocate_2():
        test_file_1 = "alllist2.txt"
        test_file_2 = "fclist.txt"
        test_file_3 = "deptlist.txt"

        S_test = readStdnts(test_file_1)
        F_test = readFreeChoice(test_file_2)
        C_test = readDeptCapacity(test_file_3)

        test_2_result = {'civil':[], 'chemical':[], 'electrical':[], 'mechanical':[], 'software':[],
            'materials':[], 'engphys':[]}
        test_2_actual = allocate(S_test ,F_test ,C_test)
        result = test_2_result == test_2_actual
        assertionEqual("allocate(S,F,C)", True, result , "Empty student file allocation")


def test():
        test_average_1()
        test_average_2()
        test_average_3()
        test_sort_1()
        test_sort_2()
        test_allocate_1()
        test_allocate_2()

test()
```

I Code for Partner's CalcModule.py

```
## @file CalcModule.py
#  @author kuber khanna
#  @brief Performs necessary operations on student data such as sorting on basis of `gpa`, `average
     gpa calculation` and appropriate `allocation` of students into their respective departments
#  @date 17/01/2019

from operator import itemgetter
from ReadAllocationData import *
##  @brief sort sorts student data according to their gpa in descending order
#   @param S list of student data generated by readStdnts in the format ``[{'macid': string, 'fname':
     string, 'lname': string, 'gender':string, 'gpa': float, 'choices': [string, string, string],
     .....}]``
#   @return list1 : list of students sorted according to thier gpa where each element of the list has
     the form ```{'macid': string, 'fname': string, 'lname': string, 'gender':string, 'gpa': float,
     'choices': [string, string, string]}```

#IMPLEMENTATION OF THIS FUNCTION IS INSPIRED FROM THE FOLLOWING LINK:
     https://www.geeksforgeeks.org/ways-sort-list-dictionaries-values-python-using-itemgetter/
def sort(S):
    if S == []:
        return ("Data is empty")
    if type(S) != list:
        return("type error")
    else:
            list1 = []
            list1 = sorted(S, key = itemgetter('gpa') , reverse = True)
            #print(list1)
            return list1
## @brief average calculates average gpa of students from the student data based on their gender
#  @param L list of student data generated by readStdnts where each element of the list has the form
     ``{'macid': string, 'fname': string, 'lname': string, 'gender':string, 'gpa': float, 'choices':
     [string, string, string]}``
#  @param g gender either `male` or `female`
#  @return avg : average based on the gender
def average(L , g):
    avg=0
    if type(L) != list:
        return("type error")

    if g != 'male' and g != "female":
        return("incorrect arguement for g")

    if L == []:
        return("Data is empty")

    else:
            list1 = []
            list2 = []
            for i in L:
                    if(i['gender'] == g):
                            list1.append(i)
                            for j in list1:
                                    list2.append(j["gpa"])
                            avg=sum(list2)//len(list2)
            #print(avg)
    return(avg)
##  @brief performs necessary operations to allocate students to their preffered department based on
     their `gpa` or whether thay have `free choice`
#  @param S list of student data generated by readStdnts where each entry of the list has the form
     ``{'macid': string, 'fname': string, 'lname': string, 'gender':string, 'gpa': float, 'choices':
     [string, string, string]}``
#  @param F list of students with free choice generated by function readFreeChoice
#  @param C dictionary of the form `{department : capacity}` returned by readDeptCapacity
#  @return final_dict : dictionary  of form ```{department : [student, .... student]}``` where student
     is the same format returned by readStdnts
def allocate(S,F,C):
    if (S == [] or F == [] or C == {}):
        return ("Data is empty")
    if (type(S) != list or type(F)!= list or type(C)!=dict):
        return ("type error")
    else:
          final_dict = {}

          passed = []
          probation = []
```

```
    for x in S:
            if x['gpa'] >= 4.0:
                    passed.append(x)
            else:
                    probation.append(x)

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

    for x in passed:
            if x['macid'] in F and C[x['choices'][0]] > 0:
                    allocated[x['choices'][0]].append(x)
                    C[x['choices'][0]] -= 1
            else:
                    sortByGPA.append(x)

    sortByGPA.sort(key=itemgetter('gpa'), reverse=True)

    deptFull = []

    for x in sortByGPA:

            if C[x['choices'][0]] > 0:
                    allocated[x['choices'][0]].append(x)
                    C[x['choices'][0]] -= 1

            elif C[x['choices'][1]] > 0:
                    allocated[x['choices'][1]].append(x)
                    C[x['choices'][1]] -= 1

            elif C[x['choices'][2]] > 0:
                    allocated[x['choices'][2]].append(x)
                    C[x['choices'][2]] -= 1
            else:
                    deptFull.append(x)

  # Students Who Were Successfully Allocated To Their Choice(s)
    for x in allocated:
          #print(x)
          #print(allocated[x])
          #print()

  # Students Who Had A GPA Greater Than 4, But All Departments Were Full
  # print(deptFull)
  # print()

  # # Students With A GPA Less Than 4.0
  # print(probation)
  # print()
     final_dict = {'civil' : allocated['civil'] , 'chemical' : allocated['chemical'] ,
         'electrical' : allocated['electrical'] , 'mechanical' : allocated['mechanical'] ,
         'software' : allocated['software'] , 'materials' : allocated['materials'] , 'engphys'
         : allocated['engphys']}


#          print(final_dict)
     return final_dict
```

J Makefile

```
PY = python3
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
```