Assignment 2 Solution

Raymond Tu tur1 400137262

February 24, 2019

Report for 2AA4 Assignment 2.

1 Testing of the Original Program

1.1 Rationale for Test Case Selection

For test case selection I aimed to have as much coverage of the modules as possible, which is to test every function and it's possibilities. Due to the nature of the specification for the modules and their functions, this meant that usually functions in every module had 2 test cases, 1 as the base case which usually consisted of a standard input and expecting a standard output as in the specification, and an exception case where the test would check whether the function correctly raises it's python exception error.

1.2 Summary of Results

Overall, there were 25 test cases for testing the 3 modules in the specification. In the final submission, all 25 test cases passed, however in my own individual testing process there were failed test cases that will be elaborated on next. For coverage, SeqADT and DCapLst both had 100% coverage, while SALst had 78% coverage.

1.3 Problems uncovered through testing

Notably, in my own testing process I discovered errors in the logic for raising the exceptions in DCapLst. The main error in these test cases was that I was incorrectly raising the exception so it would be called every time, when put outside of the for loop this error was resolved. An additional error was an error in translation between specification and python code, where SeqADT.next() function was incorrectly outputting the next index, this error was uncovered when testing the allocate function in SALst. The error was

incorrectly returning the bounds, with s[i] instead of s[i-1], when these errors were fixed all test cases ran correctly.

2 Results of Testing Partner's Code

Due to how I structured my unit testing module, I had to combine my AALst and StdnAllocTypes files with my partner files in order for my unit testing to correctly run. This is because in my test cases for SALst, I imported these 2 modules to have more freedom in the test cases for allocate and sort, to have better coverage and a more comprehensive unit testing module overall. Out of the 25 test cases, 9 failed, 3 cases in SeqADT, 3 cases in DCapLst and 3 cases in SALst.

2.1 Errors in SeqADT

In SeqADT, the test cases for init, start and end failed. For init and start, this was only an error in the differences between the named variables, while my partner added the dunders around their variable names, the test case followed the specification with variable names s and i instead of __s and __i. The end function likely failed to return the correct boolean due to the errors in the next function, which the test case relied on to test end().

2.2 Errors in DCapLst

The failed test cases in DCapLst were in the init, add and remove functions. These were mainly due to the differences in our implementation of the specification for the set of tuples, while I used a set of tuples defined in the module, my partner used a list of tuples. When this was changed accordingly the test cases passed.

2.3 Errors in SALst

The failed test cases in SALst were in the init, remove and info functions. The errors were similar to the ones in the DCapLst, being a difference in our implementation of the specification. The partner files used a list of tuples while the test cases expected a set of tuples, when this error was fixed the test cases passed.

3 Critique of Given Design Specification

In comparison to assignment 1, this time we got a formal design specification to follow, with various advantages and drawbacks. With a formal design specification, there is the benefit that the errors when testing other people's code are easily found, and usually as a result of a different implementation of the specification. Additionally, there is less room for assumptions or interpretation, the main difficulty with this assignment was understanding the specification. While it was relatively easy to understand for most modules, more complicated functions such as the sort and average function were difficult to parse through the nested quantifiers in the specification. Improvements that could be made would be to provide both a natural language specification and a formal design specification, so there is increased clarity on the implementation for the given problem.

4 Answers

1. In a natural language specification, such as the first assignment the benefit is that it is very easy to understand the problem and implement a solution. However, the problems with it are that many things are left open to assumption, leading to differing code between every programmer. Additionally, how to implement each solution are usually left open to interpretation. In a formal language specification, thing such as the inputs, outputs, format of the module and solution are very clear, there is little need to understand the context behind the problem but instead following the specification. However, there is difficulty with parsing a formal specification in comparison to a natural language specification, and working with a formal specification assumes the programmer is familiar with the format. Overall, while a formal specification leads to more standardized code, ideally both would be given for an assigned problem.

2. The specification would need to be modified when reading the input data, if the given input has a GPA outside of the bounds between 0 and 12 the exception would be raised. While the specification could be modified to replace a record type with a new ADT it is likely not necessary, having the exception in the case of reading the data to the program would ensure the program would not work with these invalid values. Because of the modularity of the program, there are multiple methods to implement an assumption as a requirement.

3. The documentation could be modified so that DCapALst is made to be a generic ADT, so it can take all types of values. Then for SALst these functions could be imported and used, reducing the amount of work and code needed for these modules to take advantage of the similarities. For example, because of the similarity of these 2 modules, code was able to be copied and modified between these modules due to the similarity of their functions. Functions such as add or elm if specified with a general type T could then be formally defined once to avoid repetition in the specification.

4. A2 is more general than A1 due to the formal specification. Because the formal specification merely gives us the specification for the modules to implement, without any of the context for it's purpose. This means that potentially the modules from A2 could be used for other purposes other than student allocation, because of this generality it has more potential function. It is because of this lack of generality that there was more ambiguity in the implementation of the modules in A1, requiring large amounts of assumptions that often differed between programmers. A natural language specification such as in A1, while being easier to understand, is less general because these assumptions mean that the modules would only work for one very specific interpretation of the problem.

5. The advantages over using SeqADT over a python list is that there is more freedom in defining additional functions to manipulate the list in SeqADT in comparison to a regular python list. An example of this is the end function that returns a boolean over whether the end of the list is reached. With SeqADT, the iteration over the list and index can be easily tracked, while for a regular python list this has to be manually tracked with a counter, which can lead to more complex and less readable code. Additionally, with SeqADT there is the opportunity for encapsulation of the state variables s and i. Each of the functions in SeqADT to access the variables are able to be controlled, so if desired it could be modified to provide more encapsulation from the user if the program is sensitive.

6. Enums prevent some of the problems in A1 such as misspelling or differing capitalization for the given data types such as gender. This greatly reduces the risk of accidentally changing the name of the data type, and further improves the similarity between the code with our peers. Enums are not introduced in the specification for macids because enums have a set name, such as male or female, while macids are intended to vary for every student, so it would not make sense to use an enum for macids.

E Code for StdntAllocTypes.py

```
## @file StdntAllocTypes.py
#  @title StdntAllocTypes
#  @author Raymond Tu
#  @date 11/02/2019
from SeqADT import *
from enum import Enum
from typing import NamedTuple


## @brief This class represents the gender type enum.
#  @details This class has num values for male and female.
class GenT(Enum):
    male = 0
    female = 1


## @brief This class represents the department type enum.
#  @details This class has num values for 7 engineering departments.
class DeptT(Enum):
    civil = 1
    chemical = 2
    electrical = 3
    mechanical = 3
    software = 4
    materials = 5
    engphys = 6


## @brief This class represents the student info of type named tuple.
#  @details This class has fields that denote information for a student
#  including first name, gender, GPA, program choices and free choice.
class SInfoT(NamedTuple):
    fname: str
    lname: str
    gender: GenT
    gpa: float
    choices: SeqADT
    freechoice: bool
```

F Code for SeqADT.py

```
## @file SeqADT.py
#  @title SeqADT
#  @author Raymond Tu
#  @date 11/02/2019

## @brief This class represents a generic abstract object.
#  @details This class iterates through a sequence of generic type T.


class SeqADT:

    ## @brief Constructor for SeqADT.
    #  @details Constructor accepts x, making it equal to sequence s,
    #  initializes integer i.
    #  @param x being the object passed into SeqADT, of any generic type.
    def __init__(self, x):
        self.s = x
        self.i = 0

    ## @brief Initializes integer i to 0.
    def start(self):
        self.i = 0

    ## @brief Iterates sequences SeqADT.
    #  @details Iterates the index i to the next value in the sequence.
    #  @return Returns the value at s[i].
    def next(self):
        if (self.i > len(self.s)) or (self.i == len(self.s)):
            raise StopIteration

        self.i = self.i + 1
        return self.s[self.i - 1]

    ## @brief Checks if at end of sequence s.
    #  @details Checks whether index has reached end of sequence.
    #  @return Returns boolean for whether the end of the list is reached.
    def end(self):
        return (self.i >= len(self.s) - 1)
```

G Code for DCapALst.py

```
## @file DCapALst.py
#  @title DCapALst
#  @author Raymond Tu
#  @date 11/02/2019
import collections


## @brief This class represents the list of department capacities.
#  @details This class holds the departments in engineering and their capacities.
class DCapALst:

    s = set()

    ## @brief Constructor for DCapALst.
    #  @details Constructor initializes the set.
    @staticmethod
    def init():
        DCapALst.s = set()
    ## @brief Adds department to set of department capacities.
    #  @details Takes in the department as well as it's capacity.
    #  @param d being the engineering department.
    #  @param n being the capacity of the department.
    @staticmethod
    def add(d, n):
        for tuple in DCapALst.s:
            if tuple.deptT == d:
                raise KeyError

        newTuple = collections.namedtuple('newTuple', ['deptT', 'cap'])
        addtuple = newTuple(d, n)

        DCapALst.s.add(addtuple)

    ## @brief Removes department from set of department capacities.
    #  @details Takes in the department to remove.
    #  @param d being the engineering department.
    @staticmethod
    def remove(d):
        for tuple in DCapALst.s:
            if tuple.deptT == d:
                DCapALst.s.remove(tuple)
                return
        raise KeyError

    ## @brief Checks if the department is in the set.
    #  @details Takes in the department.
    #  @param d being the engineering department.
    #  @return Returns a boolean, True if the department is in the set, false otherwise.
    @staticmethod
    def elm(d):
        for tuple in DCapALst.s:
            if tuple.deptT == d:
                return True
    ## @brief Checks the department capacity.
    #  @details Takes in the department and checks capacity.
    #  @param d being the engineering department.
    #  @return Returns the capacity of the passed in department.
    @staticmethod
    def capacity(d):
        for tuple in DCapALst.s:
            if tuple.deptT == d:
                return tuple.cap
        raise KeyError
```

H Code for AALst.py

```
## @file AALst.py
#  @title AALst
#  @author Raymond Tu
#  @date 11/02/2019
from StdntAllocTypes import *

## @brief This class represents the allocated list of students.
#  @details This class holds the list of students who are allocated for second year.


class AALst:

    s = []

    ## @brief Constructor for AALst.
    #  @details Constructor initializes the sequence with all departments,
    #  the set is of AllocAssocListT.
    @staticmethod
    def init():
        AALst.s = [{d: []} for d in DeptT]

    ## @brief Adds student to sequence of AllocAssocListT.
    #  @details Takes in the deptT as well as the string for student.
    #  @param dep being the department of the student.
    #  @param m being the string that represents the student.
    @staticmethod
    def add_stdnt(dep, m):
        for tuple in AALst.s:
            for d in tuple:
                if d == dep:
                    tuple[d].append(m)
    ## @brief Goes through list of allocated students and returns their macid.
    #  @details Takes in the department and returns sequence of students enrolled.
    #  @param d represents the department name.
    #  @return Returns list of students macids enrolled in that department.
    @staticmethod
    def lst_alloc(d):
        for tuple in AALst.s:
            for key in tuple:
                if key == d:
                    return tuple[key]
    ## @brief Goes through list of allocated students and returns number of students
        enrolled.
    #  @details Takes in the department and returns number of students enrolled.
    #  @param d represents the department name.
    #  @return Returns number of students macids enrolled in that department.
    @staticmethod
    def num_alloc(d):
        for tuple in AALst.s:
            for key in tuple:
                if key == d:
                    return len(tuple[key])
```

I Code for SALst.py

```
## @file SALst.py
#  @title SALst
#  @author Raymond Tu
#  @date 11/02/2019
from StdntAllocTypes import *
from AALst import *
from DCapALst import *
import collections


## @brief This class represents the list of students and their information.
#  @details This class holds the list of all students to be allocated and their info.
class SALst:
    ## @brief Initializes the set of students.
    #  @details Initializes the set of students that will contain their information.
    @staticmethod
    def init():
        SALst.s = set()

    ## @brief Adds student to list of students.
    #  @details Takes in the macid and student information.
    #  @param m being the macid.
    #  @param i being the tuple of student information.
    @staticmethod
    def add(m, i):
        for tuple in SALst.s:
            if tuple.macid == m:
                raise KeyError
        StudentT = collections.namedtuple('StudentT', ['macid', 'info'])
        new_stdnt = StudentT(m, i)
        SALst.s.add(new_stdnt)
    ## @brief Removes student from list of students.
    #  @details Takes in the macid of the student to be removed.
    #  @param m being the macid.
    @staticmethod
    def remove(m):
        for tuple in SALst.s:
            if tuple.macid == m:
                SALst.s.remove(tuple)
                return
        raise KeyError

    ## @brief Checks if student is in the list of students.
    #  @details Takes in macid and checks if they've been added.
    #  @param m being the macid.
    #  @return True Returns boolean for whether student has been added.
    @staticmethod
    def elm(m):
        for tuple in SALst.s:
            if tuple.macid == m:
                return True
    ## @brief Gives the information for the passed in student.
    #  @details Returns information of student associated with macid.
    #  @param m being the macid.
    #  @return info Returns the information of the macid.
    @staticmethod
    def info(m):
        for tuple in SALst.s:
            if tuple.macid == m:
                return tuple.info
        raise ValueError

    # Worked with Lucia Cristiano and Charles Zhang
    ## @brief Sorts the list of students by GPA in descending order.
    #  @details Takes in parameters and sorts students in descending order.
    #  @param f Lambda function to determine which students are sorted.
    #  @return L Returns list of sorted students.
    @staticmethod
    def sort(f):
        li = []
        for tuple in SALst.s:
            if f(tuple.info):
                li.append(tuple.macid)

        macid = 0
        next_macid = 0
```

```
        for macid in range(len(li)):
            for next_macid in range(0, (len(li)) - next_macid - 1):
                if SALst.__get_gpa(li[next_macid], SALst.s) <
                    SALst.__get_gpa(li[next_macid + 1], SALst.s):
                    tmp = li[next_macid]
                    li[next_macid] = li[next_macid + 1]
                    li[next_macid + 1] = tmp

        return li

    # Worked with Lucia Cristiano and Charles Zhang
    ## @brief Computes the average GPA of all students.
    #  @details Returns average GPA of all students in list.
    #  @param f Being the lambda function that determines which students to average.
    #  @return avg Returns the average of the students.
    @staticmethod
    def average(f):
        fset = [x[1] for x in SALst.s if f(x[1])]
        if fset == []:
            raise ValueError
        print(fset)
        fset_sum = 0
        for sinfo in fset:
            fset_sum = fset_sum + sinfo[3]
        return fset_sum / (len(fset))

    ## @brief Allocates the students into the AALst.
    #  @details Allocates the students into the AALst based on gpa and freechoice.
    @staticmethod
    def allocate():
        AALst.init()
        f = SALst.sort(lambda t: t.freechoice and t.gpa >= 4.0)
        for m in f:
            ch = SALst.info(m).choices
            AALst.add_stdnt(ch.next(), m)
        s = SALst.sort(lambda t: (not t.freechoice) and t.gpa >= 4.0)
        for m in s:
            ch = SALst.info(m).choices
            alloc = False
            while (not alloc) and (not ch.end()):
                d = ch.next()
                if AALst.num_alloc(d) < DCapALst.capacity(d):
                    AALst.add_stdnt(d, m)
                    alloc = True
            if not alloc:
                raise RuntimeError

    @staticmethod
    def __get_gpa(m, s):
        for tuple in s:
            if tuple.macid == m:
                return tuple.info.gpa
```

J Code for Read.py

```
## @file Read.py
#  @title Read
#  @author Raymond Tu
#  @date 11/02/2019
from StdntAllocTypes import *
from DCapALst import *
from SALst import *


## @brief Reads textfile of students and puts them into the list of students.
#  @details Takes formatted student list and converts them to list of student tuples.
#  @param s The passed in textfile.
def load_stdnt_data(s):
    SALst.init()
    file = open(s, 'r')
    lines = file.readlines()

    for line in lines:
        info_list = line.split(', ', 5)

        # get list of choices
        choice_split = info_list[-1].split('], ')
        choice_split1 = choice_split[0].split('[')
        choices_str = choice_split1[1]
        choices = choices_str.split(', ')

        f_c = True
        # convert true false to bool
        if choice_split[1] == 'True':
            f_c = True
        elif choice_split[1] == 'False':
            f_c = False

        for choice in choices:
            if choice == 'civil':
                choice = DeptT.civil
            elif choice == 'chemical':
                choice = DeptT.chemical
            elif choice == 'electrical':
                choice = DeptT.electrical
            elif choice == 'mechanical':
                choice = DeptT.mechanical
            elif choice == 'software':
                choice = DeptT.software
            elif choice == 'materials':
                choice = DeptT.materials
            elif choice == 'engphys':
                choice = DeptT.engphys

        choices = SeqADT(choices)

        if info_list[3] == 'male':
            info_list[3] = GenT.male
        if info_list[3] == 'female':
            info_list[3] == GenT.female
        info_list[4] = float(info_list[4])
        sinfo = SInfoT(info_list[1], info_list[2], info_list[3], info_list[4], choices,
            f_c)
        SALst.add(info_list[0], sinfo)


## @brief Reads textfile of departments and puts them into the list of departments.
#  @details Takes list of departments and converts them to list of department tuples.
#  @param s The passed in textfile.
def load_dcap_data(s):
    DCapALst.init()
    file = open(s, 'r')
    lines = file.readlines()
    for line in lines:
        dept_list = line.split(', ')

        if dept_list[0] == 'civil':
            dept_list[0] = DeptT.civil
        elif dept_list[0] == 'chemical':
            dept_list[0] = DeptT.chemical
        elif dept_list[0] == 'electrical':
```

```
        dept_list[0] = DeptT.electrical
    elif dept_list[0] == 'mechanical':
        dept_list[0] = DeptT.mechanical
    elif dept_list[0] == 'software':
        dept_list[0] = DeptT.software
    elif dept_list[0] == 'materials':
        dept_list[0] = DeptT.materials
    elif dept_list[0] == 'engphys':
        dept_list[0] = DeptT.engphys

    dept_list[1] = int(dept_list[1])
    DCapALst.add(dept_list[0], dept_list[1])
```

K Code for test_All.py

```
import pytest
from SeqADT import *
from DCapALst import *
from SALst import *
from AALst import *


class TestAll:

    # SeqADT tests
    def test_seqadt_init(self):
        new_seq = SeqADT([0, 1, 2, 3])
        assert new_seq.s == [0, 1, 2, 3]

    def test_seqadt_start(self):
        new_seq = SeqADT([])
        new_seq.start()
        assert new_seq.i == 0

    def test_seqadt_next_base(self):
        new_seq = SeqADT([0, 1, 2, 3])
        index = new_seq.next()
        assert index == 0

    def test_seqadt_next_excep(self):
        new_seq = SeqADT([])
        with pytest.raises(StopIteration):
            index = new_seq.next()

    def test_seqadt_end(self):
        new_seq = SeqADT([0, 1, 2, 3])
        new_seq.next()
        new_seq.next()
        new_seq.next()
        end_bool = new_seq.end()
        assert end_bool is True

    # DCapALst tests
    def test_dcapalst_init(self):
        DCapALst.init()
        empty_set = set()
        assert DCapALst.s == empty_set

    def test_dcapalst_add_base(self):
        DCapALst.init()
        DCapALst.add(DeptT.civil, 2)
        print(DCapALst.s)
        for tuple in DCapALst.s:
            dep = tuple.deptT
            cap = tuple.cap
        assert (dep == DeptT.civil) and (cap == 2)

    def test_dcapalst_add_excep(self):
        DCapALst.init()
        DCapALst.add(DeptT.civil, 2)
        with pytest.raises(KeyError):
            DCapALst.add(DeptT.civil, 2)

    def test_dcapalst_remove_base(self):
        DCapALst.init()
        DCapALst.add(DeptT.civil, 2)
        DCapALst.remove(DeptT.civil)
        empty_set = set()
        assert DCapALst.s == empty_set

    def test_dcapalst_remove_excep(self):
        DCapALst.init()
        print(DCapALst.s)
        with pytest.raises(KeyError):
            DCapALst.remove(DeptT.civil)

    def test_dcapalst_elm(self):
        DCapALst.init()
        DCapALst.add(DeptT.civil, 2)
        assert DCapALst.elm(DeptT.civil) is True
```

```
def test_dcapalst_capacity_base(self):
    DCapALst.init()
    DCapALst.add(DeptT.civil, 2)
    assert DCapALst.capacity(DeptT.civil) == 2

def test_dcapalst_capacity_excep(self):
    DCapALst.init()
    with pytest.raises(KeyError):
        DCapALst.capacity(DeptT.civil)

# SALst test cases

def test_salst_init(self):
    SALst.init()
    empty_set = set()
    assert SALst.s == empty_set

def test_salst_add_base(self):
    SALst.init()
    sinfo1 = SInfoT("first", "last", GenT.male, 12.0, SeqADT([DeptT.chemical]), True)
    SALst.add("stdnt1", sinfo1)
    for stdnt in SALst.s:
        gender = stdnt[1].gender
        gpa = stdnt[1].gpa
    assert (gender == GenT.male) and (gpa == 12.0)

def test_salst_add_excep(self):
    SALst.init()
    sinfo1 = SInfoT("first", "last", GenT.male, 12.0, SeqADT([DeptT.chemical]), True)
    SALst.add("stdnt1", sinfo1)
    with pytest.raises(KeyError):
        SALst.add("stdnt1", sinfo1)

def test_salst_remove_base(self):
    SALst.init()
    sinfo1 = SInfoT("first", "last", GenT.male, 12.0, SeqADT([DeptT.chemical]), True)
    SALst.add("stdnt1", sinfo1)
    SALst.remove("stdnt1")
    empty_set = set()
    assert SALst.s == empty_set

def test_salst_remove_excep(self):
    SALst.init()
    with pytest.raises(KeyError):
        SALst.remove("stdnt1")

def test_salst_elm(self):
    SALst.init()
    sinfo1 = SInfoT("first", "last", GenT.male, 12.0, SeqADT([DeptT.chemical]), True)
    SALst.add("stdnt1", sinfo1)
    assert SALst.elm("stdnt1") is True

def test_salst_info_base(self):
    SALst.init()
    sinfo1 = SInfoT("first", "last", GenT.male, 12.0, SeqADT([DeptT.chemical]), True)
    SALst.add("stdnt1", sinfo1)
    assert SALst.info("stdnt1") == sinfo1

def test_salst_info_excep(self):
    SALst.init()
    with pytest.raises(ValueError):
        SALst.info("stdnt1")

def test_salst_average_base(self):
    SALst.init()
    sinfo1 = SInfoT("first", "last", GenT.male, 12.0, SeqADT([DeptT.civil]), True)
    SALst.add("stdnt1", sinfo1)
    avg = SALst.average(lambda x: x.gender == GenT.male)
    assert avg == 12.0

def test_salst_average_excep(self):
    SALst.init()
    with pytest.raises(ValueError):
        SALst.average(lambda x: x.gender == GenT.male)

def test_salst_sort(self):
    SALst.init()
    sinfo1 = SInfoT("first", "last", GenT.male, 12.0, SeqADT([DeptT.chemical]), True)
    SALst.add("stdnt1", sinfo1)
    sort_list = SALst.sort(lambda t: t.freechoice and t.gpa >= 4.0)
```

```
    assert sort_list == ['stdnt1']

def test_salst_allocate(self):
    SALst.init()
    sinfo1 = SInfoT("first", "last", GenT.male, 12.0, SeqADT([DeptT.civil]), True)
    SALst.add("stdnt1", sinfo1)
    AALst.init()
    SALst.allocate()
    alloc = AALst.lst_alloc(DeptT.civil)
    assert alloc == ['stdnt1']
```

L Code for Partner's SeqADT.py

```
## @file SeqADT.py
#  @author janzej2
#  @brief Implementation of an abstract data type for a sequence.
#  @date 02/11/19


## @brief This class represents an abstract data type of type sequence.
#  @details This class represents a sequence initialised with a list s and a count
#  variable i.
class SeqADT:

    ## @brief Constructor for SeqADT
    #  @details Constructor accepts one parameter consisting of a list of a type T
    #   and initialises its parameter i to 0.
    #  @param s List containing a specific data type.
    def __init__(self, s):
        self.__s = s
        self.__i = 0

    ## @brief Initialiser for i variable, resetting it to its original 0 value.
    def start(self):
        self.__i = 0

    ## @brief Function to iterate through the sequence and return the most recent value.
    #  @return Returns the i-th element before it is incremented.
    def next(self):
        if self.end():
            raise StopIteration
        self.__i += 1
        return self.__s[self.__i - 1]

    ## @brief Function to determine if the end of the sequence has been reached.
    #  @return Returns True if the function has reached the end, or False otherwise.
    def end(self):
        if self.__i >= len(self.__s):
            return True
        return False
```

M Code for Partner's DCapALst.py

```
## @file DCapALst.py
#  @author janzej2
#  @brief A module which defines the abstract data type for various departments
#  @date 02/10/19

# from StdntAllocTypes import *


## @brief This class represents an abstract data type of department capacities.
#  @details Initialising this object creates a new tuple used to store departments
#  and their relevant capacities.
class DCapALst:
    s = []
    ## @brief Initialises an empty tuple.
    @staticmethod
    def init():
        DCapALst.s = []

    ## @brief Adds a department and its capacity to the DCapALst.
    #  @param d Represents a department of type DeptT.
    #  @param n Represents the capacity of that department.
    @staticmethod
    def add(d, n):
        for dept in DCapALst.s:
            if dept[0] == d:
                raise KeyError
        DCapALst.s.append([d, n])

    ## @brief Removes a given department from the DCapALst.
    #  @param d Represents the DeptT to remove from the list.
    @staticmethod
    def remove(d):
        removed = False
        for dept in DCapALst.s:
            if dept[0] == d:
                DCapALst.s.remove(dept)
                removed = True
        if not removed:
            raise KeyError

    ## @brief Checks if a given department is already in the list.
    #  @param d Represents the DeptT to check for in the list.
    #  @return Returns a boolean value representing whether the
    #   department is in the list.
    @staticmethod
    def elm(d):
        for dept in DCapALst.s:
            if dept[0] == d:
                return True
        return False

    ## @brief Checks the capacity of a given department.
    #  @param d Represents the DeptT to check.
    #  @return Returns the capacity of the given department if
    #   it is in the list.
    @staticmethod
    def capacity(d):
        for dept in DCapALst.s:
            if dept[0] == d:
                return dept[1]
        raise KeyError
```

N Code for Partner's SALst.py

```
## @file SALst.py
#  @author janzej2
#  @brief A module which defines the behvaiour of the Student Association List.
#  @date 02/11/19
#  shared ideas with Ahmed Al Koasmh and David Thompson

from StdntAllocTypes import *
from AALst import *
from DCapALst import *


## @brief A class representing the data type SALst.
#  @details This module contains function for initialising a new list,
#  adding a student to the list (given their MacID and student information),
#  removing a student from the list, determining if a student is in the list
#  by MacID, returning a student's student information by MacID, sorting a
#  group of students by a given f, taking the average of a set of students
#  based on a condition f, and a function to allocate students to their programs.
class SALst:

    s = []

    ## @brief Initialises the empty student allocation list.
    def init():
        SALst.s = []

    ## @brief Adds a student to the student allocation list.
    #  @param m A string representing the student's MacID.
    #  @param i A SInfoT object representing a student's info.
    @staticmethod
    def add(m, i):
        for element in SALst.s:
            if element[0] == m:
                raise KeyError
        SALst.s.append([m, i])

    ## @brief Removes a student from the student allocation list.
    #  @param m A string representing a student's MacID.
    @staticmethod
    def remove(m):
        removed = False
        for element in SALst.s:
            if element[0] == m:
                SALst.s.remove(element)
                removed = True
        if not removed:
            raise KeyError

    ## @brief Determines if a student is an element of the list.
    #  @param m A string representing a student's MacID.
    #  @return True if the student is in the list, and false otherwise.
    @staticmethod
    def elm(m):
        for element in SALst.s:
            if element[0] == m:
                return True
        return False

    ## @brief Checks a student's MacID and returns their student information.
    #  @param m A string representing a student's MacID.
    #  @return The student's information as type SInfoT.
    @staticmethod
    def info(m):
        for element in SALst.s:
            if element[0] == m:
                return element[1]
        raise KeyError

    ## @brief Sorts the SALst based on parameters given through f.
    #  @param f Represents a function to determine the method of sorting
    #  @return Returns a list of MacIDs ordered based on the input condition.
    @staticmethod
    def sort(f):
        sorted_list = sorted(SALst.s, key=lambda i: i[1].gpa, reverse=True)
        return [item[0] for item in sorted_list if f(item[1])]
```

```
## @brief Returns the average of a group of students based on the input criteria.
#  @param f Represents a function to filter based on.
#  @return Returns the average of the students that match the given criteria.
@staticmethod
def average(f):
    avg_list = [item[1].gpa for item in SALst.s if f(item[1])]
    if len(avg_list) == 0:
        raise ValueError
    sum = 0
    for item in avg_list:
        sum += item
    return sum / len(avg_list)

## @brief Allocates students based on their student information.
#  @details Creates a list of freechoice students first, allocating
#  them to their first choice program. Then, the function sorts the student
#  list by GPAs >= 4.0, and allocates them to their first, second or third
#  choice programs based on remaining space.
@staticmethod
def allocate():
    AALst.init()
    f = SALst.sort(lambda t: t.freechoice and t.gpa >= 4.0)
    for m in f:
        ch = SALst.info(m).choices
        AALst.add_stdnt(ch.next(), m)

    s = SALst.sort(lambda l: not l.freechoice and l.gpa >= 4.0)
    for m in s:
        ch = SALst.info(m).choices
        alloc = False
        while not alloc and not ch.end():
            d = ch.next()
            if AALst.num_alloc(d) < DCapALst.capacity(d):
                AALst.add_stdnt(d, m)
                alloc = True
        if not alloc:
            raise RuntimeError
```