Assignment 2 Solution

Your name here

February 16, 2020

Brief This report is a write-up and analysis of the module interface specification (MIS) for Assignment 2. It discusses and critiques the intricate details of my code and partner's based upon the test cases that I made. Furthermore, it repeats this analytical process for the MIS and discusses the restrictions, problems, and shortcomings with the MIS. In addition, potential solutions are documented. Finally, it answers questions outlined in the MIS. The program was written in Python programming language with the code for both my own program and my partners program given in the pages below.

1 Testing of the Original Program

Tests for the program was set up based on boundary cases and normal cases. Extreme cases were not considered as the assumed chemical reactions are deemed not to have extreme or egde cases. Pytest was used for testing the modules.

Testing for TestSet:

- `test_add`: Test for the method add in Set. This test adds two elements to the list, one of type integer and one of type string. Due to python having dynamic typing, both are accepted as a part of the set and added to it.
- `test_rm`: Test for the method rm in Set. This test removes a member from a set and checks if it is still present. It also attempts to remove a member not present but runs into ValueError as expected.
- `test_member`: Test for the method member in Set. This test checks for two members in the set, one present and one not present.
- `test_size`: Test for the method size in Set. This test checks the size of the two sets: One a set that is a normal set with 5 elements and another which is an empty set of size 0.

- `test_equals`: Test for the method equals in Set. This test checks equality for sets containing the same elements. And checks if a subset of the set is deemed equal to the set.
- `test__eq__`: Test for the method `__eq__` in Set. This test checks equality for sets containing the same elements. And checks if a subset of the set is deemed equal to the set.
- `test_to_seq`: Test for the method to_seq in Set. This test checks that this function returns a sequence of elements in the set as a list.

Testing for TestMoleculeT:

- `test_get_elm`: Test the method get_elm in MoleculeT. This test checks if it gets the right element from MoleculeT.
- `test_get_num`: Test the method get_num in MoleculeT. This test checks if it gets the right number of atoms of an element from MoleculeT.
- `test_equals`: Test the method equals in MoleculeT. This test checks that it is equal to a similar molecule, checks that it is not equal to a molecule with the same element but different number of atoms and, also checks that it is not equal to a molecule with same number of atoms but a different element.
- `test__eq__`: Test the method `__eq__` in MoleculeT. This test checks that it is equal to a similar molecule, checks that it is not equal to a molecule with the same element but different number of atoms and, also checks that it is not equal to a molecule with same number of atoms but a different element.
- `test_num_atoms`: Test the method num_atoms in MoleculeT. This test checks that it returns the correct number of atoms of the element in MoleculeT and returns 0 if the element is not present in the MoleculeT.
- `test_constit_elms`: Test for the method constit_elms in MoleculeT. This test checks if it returns a set of the element in the molecule.

Testing for TestCompoundT:

- `test_get_molec_set`: Test for the method get_molec_set in CompoundT. This test checks if the method returns the correct MolecSet for three compounds. These compounds consist of three, three and one molecule each as that was deemed by me the standard for compounds.

- `test_equals`: Test for the method equals in CompoundT. This test checks if the method is equal to the compounds that are identical but not equal to compounds that are slightly different in terms of the number of atoms of a certain element.
- `test__eq__`: Test for the method __eq__ in CompoundT. This test checks if the method is equal to the compounds that are identical but not equal to compounds that are slightly different in terms of the number of atoms of a certain element.
- `test_num_atoms`: Test for the method num_atoms in CompoundT. This test checks if the number of atoms of a particular element are returned the same as in the Compound, and also 0 for the elements not in the compound.
- `test_constit_elms`: Test for the method constit_elms in CompoundT. This test checks if the method returns the current set of elements in the compound.

Testing for TestReactionT:

- `test_get_lhs`: Test of the method get_lhs in ReactionT. This test checks if the correct sequence of compounds on the left-hand side are returned.
- `test_get_rhs`: Test of the method get_rhs in ReactionT. This test checks if the correct sequence of compounds on the right-hand side are returned.
- `test_get_lhs_coeff`: Test for the method get_lhs_coeff in ReactionT. This test checks if the correct sequence of the coefficient of the left-hand side is returned. The coefficients are returned in natural/ whole numbers as per the guidelines of chemistry.
- `test_get_rhs_coeff`: Test for the method get_rhs_coeff in ReactionT. This test checks if the correct sequence of the coefficient of the right-hand side is returned. The coefficients are returned in natural/ whole numbers as per the guidelines of chemistry.

Passed Cases: 21
Failed Cases: 0
Total Cases: 21

2 Results of Testing Partner's Code

Testing for TestSet:

- `test_add`: Passed
- `test_rm`: Passed
- `test_member`: Passed
- `test_size`: Passed
- `test_equals`: Passed
- `test__eq__`: Passed
- `test_to_seq`: Passed

Testing for TestMoleculeT:

- `test_get_elm`: Passed
- `test_get_num`: Passed
- `test_equals`: Passed
- `test__eq__`: Failed
- `test_num_atoms`: Passed
- `test_constit_elms`: Passed

Testing for TestCompoundT:

- `test_get_molec_set`: Passed
- `test_equals`: Passed
- `test__eq__`: Passed
- `test_num_atoms`: Passed
- `test_constit_elms`: Passed

Testing for TestReactionT:

- `test_get_lhs`: Passed
- `test_get_rhs`: Passed
- `test_get_lhs_coeff`: Failed
- `test_get_rhs_coeff`: Failed

Passed Cases: 18
Failed Cases: 3
Total Cases: 21

My partener's code failed in `test__eq__` in TestMoleculeT, `test_get_lhs_coeff` and `test_get_rhs_coeff` in test.

My partner had forgotten to include __eq__ in the MoleculeT.py, however, it might be the case that in my code I had __eq__ as an abstract method in Equality.py and he had implemented in some other way.

On the otherhand, my code returned the coefficients in natural numbers while my partner's code returned the coefficients in real numbers. Even though the ration of the coefficients were right my testing methods were not made to handle that.

`Modification to test_All.py`: In my testing I had accidentally accessed the list in the sets, which was suppose to be unaccessible. I modified it to accomodate my partner's code.

3 Critique of Given Design Specification

The design specification for this assignment was in formal specification. The advantage of using a formal specification for a MIS is that it was very easy to understand the software requirements and design. This is what I liked about the design specification of assignment 2; and formal specification as a whole. An advantage of formal specification is that it leaves very little room for assumptions and hence the types of arguments and return variable were clearly defined. This also enabled encapsulation and inheritance of different classes. The specification emphasized and encouraged modularization and separation of concerns. This as a whole was very valuable as the code became minimal and ran with efficiency. Low couplling present in all classes as they utilized methods which they either inheritated or used through calling it. I believe due to information hiding there was a not a lot of opacity. I believe the program is general due to it being very specific from it's specification.

The disadvantages to assignment 2's design specification is that it is hard to understand without proper knowledge of the mathematical symbols and definition of terms. Furthermore, due to the classes only being described using formal language, it was hard to understand what was happening within the module. Potential edge cases could not be determined as the specification was very strict with it's typing and how a Reaction would be build up.

The design specification could be improved by providing a natural language description along with the formal specification. This would enable the programmer to both get the software requirements and design without the ambiguity and with the understanding that natural language provides. A stricter return type on the coefficients of the compounds should be imposed as I believe, both my partners and my own code would fail tests due to the typing of the elements in the coefficient list.

I learned a lot from this assignment, starting from creating classes to understanding the terms required for writing software and the reasons it is necessary. I would have liked to have a proper example of edge, boundary and normal cases for the reaction. Overall, the formal specification has been proven to be better than natural language.

4 Answers

a) The advantages of the natural language approach are the simplicity in understanding the specification when natural language is used. As we communicate in natural language in our everyday lives, we are used to understanding and performing the task as told when directions are given in natural language. The disadvantages of natural language is that the use of natural languages to deal with complex problems can have issues with ambiguity, inaccuracy, and inconsistency. Since there is more than one way of interpretation when it comes to describing something in natural language, there is ambiguity as it cannot be resolved according to a rule or process with a finite number of steps. The advantages of a formal specification is that the development of a formal specification provides insights and understanding of software requirements and design. Another advantage is that it's more complete than natural language, as the formality tends to highlight any incompleteness within the specification. The disadvantages of it are that some programmers, users, and clients may not have the technical background needed to understand the formal specification.

b) The process of converting the strings to logical syntactic components called parsing. A single module called parsing could be used but keeping close to pep8 format, modularization of the code would be better.

   Create a class called parser, it would contain private methods to convert a string of chemical reaction to a logical syntactic component.

   `Method parseEquation()`
   Details: Create two private arrays for left hand side (lhs) and right-hand side(rhs). Split the string with the delimiter "+" using a while loop and store them in lhs until "=" is found. We then exit the loop and enter another loop that starts splitting after the "=" sign with the delimiter "+". The lhs and rhs are complete with the corresponding strings.

   `Method parseTerm()`
   Details: Create two list of lists/ 2D array for left hand side Compound and right-hand side compounds. If there is a white space ignore that and go to next element in the string, if the element is a capital letter start counting the position and count till it either hits a number or another capital letter or while space. Store the letter in the first array list. If the Chemical Element string is followed by a number store the number as a string or if followed by a capital letter store "1" then move on to finding the rest

of the elements.

`Method parseElement()`
Details: Now that we have separated the elements and their number of atoms from the string, we start converting it to our format. Cross-check each element in the list of lists/ 2D array and store the corresponding ElementT value. Change each number to an integer.

`Method parseReaction()`
Details: Create MolecSets respective to the number of lists in the list of lists. Now, read the list of lists/ 2D array, and make them into a Molecule by taking the Element and the following number. Store them in the Molecule Set. One list in the list of lists is a compound. A MolecSet corresponds to a compound. Make the MolecSet into a compound and store it in a new array called LHS or RHS. And the ReactionT is set for making.

`Reference`: https://www.nayuki.io/res/chemical-equation-balancer-javascript/chemical-equation-balancer.ts

c) First, we would modify Chemtypes by changing what values ElementT contains.

```
class ElementT(Enum):
    H = (1, 1.008)
    He = (2, 4.003)
    Li = (3, 6.942)
    .
    .
    Og = (118, 294)
    def __init__ (self, atomic_num, molar_mass):
            self.num = atomic_num
            self.mass = molar_mass
    def mass(self):
        return self.mass
```

In the class Chemical Entity, I would include another abstract method called get_molar_mass(). This method would take in ElementT, used ElementT.H.mass to get the molar mass and return it as a real number. For MoleculeT, it would get the molar mass using the element and multiply the number it with the number of atoms in the molecule. Hence getting the mass. For CompoundT, it would iterate though each molecule and use the function get_molar_mass of the molecule to get the mass of each molecule, have a variable which adds and stores the molar mass of each molecule and then when the iteration ends returns it.

d) The usual convention in chemistry is to have natural numbers as coefficients of the chemical equations. I actually implemented an algorithm to have natural number coefficients.
The algorithm finds the greatest common divisor (gcd) of the coefficients and then multiplies the inverse of the gcd to the coefficients. The algorithm finds the gcd by calling a function called find_gcd which takes in two parameters and, finds the gcd of it. This function iteratively called and used to find the gcd of all the entire list of coefficients and then returns it.

```
num1 = coeff[0][0]
num2 = coeff[0][1]
gcd = self.__find_gcd(num1, num2)
for i in range(len(coeff)):
    for j in range(len(coeff[0])):
        gcd = self.__find_gcd(gcd, coeff[i][j])
    coeff = (1 / gcd) * coeff

def __find_gcd(self, x, y):
    while(y):
            x, y = y, x % y
    return x
```

e) The difference between dynamic and static typing is that in dynamic typing (in a language like Python), variables must necessarily be defined before they are used. In static typing (in a language like Java), variables do not need to be defined before they're used as the variables are explicitly declared before they're used in the program. The advantages of static typing include that the IDE will easily catch many programming errors quickly when the program is executed, which will help to reduce the time spent on debugging the program. Another advantage is that the type declarations serve as automatically checked documentation, which make it easier for the programmer to understand and maintain the program. The disadvantages are that sometimes the type checker (for Java in this case) would sometimes prevent a program from running when it is suppose to run without error.

`Reference`: https://www.sitepoint.com/typing-versus-dynamic-typing/

f) [(x, y) for x in range(1, 10) for y in range(1, 10) if (x % 2 == 1) and (y % 2 == 1) and (x < y)]

g)
```
def length(listA):
    return sum(list(map(lambda x : 1, listA)))
```

h) As stated in the lectures, the software interface enables the module's clients to use the service a module provides. The implementation of the interface that provides the services offered by the module. Meaning the interface is what the end user will be seeing and working with and the implementation is what the module provides for the interface to function.

i)
  i. Abstraction: Abstraction is the process of focusing on what is important while ignoring what is irrelevant. Using the principle of abstraction, we can manage the complexity of the interface to emphasize essential characteristics and supress the implementation details. Another way to design using this principle involves separating the behaviour of software components from their implementation, as there will be unnecessary coupling if this is not followed.

  ii. Anticipation of change: Anticipation of change is the principle that future change should be anticipated and planned for. The three techniques we can use to deal with change for the design of a module's interface is configuration management, information hiding, and little languages. We can manage the configuration of the module consistently so that it can be easily modified as the software evolves. We can hide the things that are likely to change inside of the module, as well as create little languages that can be used to solve similar problems.

  iii. Generality: The principle of generality is closely related to the principle of anticipation of change. It is important in designing software that is free from unnatural restrictions and limitations. Hence to solve a more general problem than the problem at hand whenever possible. One excellent example of an unnatural restriction or limitation is the use of two digit year numbers, which has led to the "year 2000" problem: software that will garble record keeping at the turn of the century. Although the two-digit limitation appeared reasonable at the time, good software frequently survives beyond its expected lifetime.

  iv. Modularity: The principle of modularity is a specialization of the principle of separation of concerns. We can enable modularity for the design of a module's interface by considering different parts of the system separately and that the parts of the system to be considered separately from their composition. This can be done either by using modular decomposition or modular composition.

v. Separation of Concerns: Separation of concerns is the principle that different concerns should be isolated and considered separatelt. The goal is to reduce complex problems to a set of simpler problems and hence enables parallelization of effort.

`Reference`:
1) https://www.d.umn.edu/ gshute/softeng/principles.html
2) Leture slides

E Code for ChemTypes.py

```
## @file ChemTypes.py
#  @Utsharga Rozario
#  @brief Module that creates enumerated type of elements
#  @date Feb 08, 2020

from enum import Enum


## @brief an enumerate data type of elements
class ElementT(Enum):
    H = 1
    He = 2
    Li = 3
    Be = 4
    B = 5
    C = 6
    N = 7
    O = 8
    F = 9
    Ne = 10
    Na = 11
    Mg = 12
    Al = 13
    Si = 14
    P = 15
    S = 16
    Cl = 17
    Ar = 18
    K = 19
    Ca = 20
    Sc = 21
    Ti = 22
    V = 23
    Cr = 24
    Mn = 25
    Fe = 26
    Co = 27
    Ni = 28
    Cu = 29
    Zn = 30
    Ga = 31
    Ge = 32
    As = 33
    Se = 34
    Br = 35
    Kr = 36
    Rb = 37
    Sr = 38
    Y = 39
    Zr = 40
    Nb = 41
    Mo = 42
    Tc = 43
    Ru = 44
    Rh = 45
    Pd = 46
    Ag = 47
    Cd = 48
    In = 49
    Sn = 50
    Sb = 51
    Te = 52
    I = 53
    Xe = 54
    Cs = 55
    Ba = 56
    La = 57
    Ce = 58
    Pr = 59
    Nd = 60
    Pm = 61
    Sm = 62
    Eu = 63
    Gd = 64
    Tb = 65
    Dy = 66
```

```
 Ho = 67
 Er = 68
 Tm = 69
 Yb = 70
 Lu = 71
 Hf = 72
 Ta = 73
W = 74
 Re = 75
 Os = 76
 Ir = 77
 Pt = 78
 Au = 79
 Hg = 80
 Tl = 81
 Pb = 82
 Bi = 83
 Po = 84
 At = 85
 Rn = 86
 Fr = 87
 Ra = 88
 Ac = 89
 Th = 90
 Pa = 91
 U = 92
 Np = 93
 Pu = 94
 Am = 95
 Cm = 96
 Bk = 97
 Cf = 98
 Es = 99
 Fm = 100
 Md = 101
 No = 102
 Lr = 103
 Rf = 104
 Db = 105
 Sg = 106
 Bh = 107
 Hs = 108
 Mt = 109
 Ds = 110
 Rg = 111
 Cn = 112
 Nh = 113
 Fl = 114
 Mc = 115
 Lv = 116
 Ts = 117
 Og = 118
```

F Code for ChemEntity.py

```
## @file ChemEntity.py
#  @author Utsharga Rozario
#  @brief Module that creates a MoleculeT ADT
#  @date Feb 08, 2020

from abc import ABC, abstractmethod
from ChemTypes import *
from ElmSet import *


## @brief an Abstract Data Type that resemblems a chemical entity
class ChemEntity(ABC):

    ## @brief Gets the number of atoms of the parameter element in the molecule
    #  @param e ElementT type parameter
    #  @return The number atoms of the element in a chemical entity
    @abstractmethod
    def num_atoms(self, e):
        pass

    ## @brief Get the element in the molecule and returns an ElmSet of it
    #  @return ElmSet of the elements in a chemical entity
    @abstractmethod
    def constit_elems(self):
        pass
```

G Code for Equality.py

```
## @file Equality.py
#  @author Utsharga Rozairo
#  @brief Module that creates an Equality ADT and inherits ABC
#  @date Feb 08, 2020

from abc import ABC, abstractmethod


## @brief An abstract data type for the equality symbol
class Equality(ABC):

    ## @brief An abstract method that defines if the parameter and the self are equal
    #  @param other A parameter of the same type as self, and is used for the comparison
    @abstractmethod
    def __eq__(self, other):
        pass
```

H Code for Set.py

```
## @file Set.py
#  @author Utsharga Rozario
#  @brief Module that creates a Set ADT
#  @date Feb 08, 2020

from Equality import *


## @brief An abstract data type that represents a set
class Set(Equality):

    ## @brief Set Constructor
    #  @details Initializes a Set of objects
    #  @param S A List of objects
    def __init__(self, s):
        self.S = s

    ## @brief Checks if both the Sets are equal
    #  @details Checks if the current set and the parameter set are both of the same length
    #           and then if elements in both the sets are equal
    #  @param other A Set ADT
    #  @return True if the length and the elements are the same, False otherwise
    def __eq__(self, other):
        if (len(other.S) == len(self.S)):
            for e in self.S:
                if not other.member(e):
                    return False
            return True
        return False

    ## @brief Adds an object to the list
    #  @param e an element data to be added
    def add(self, e):
        self.S.append(e)

    ## @brief Removes an element from the set
    #  @details Uses the member function to determine if the element is in the list
    #           then removes the element if present
    #  @param e an element data to be removed
    #  @throw ValueError if the element is not present in the list hence cannot be removed
    def rm(self, e):
        self.S.remove(e)

    ## @brief Checks if the element in the Set
    #  @param e an element data
    #  @return True if e is present in list S
    def member(self, e):
        if e in self.S:
            return True
        return False

    ## @brief Returns the size of the list
    #  @return The length of the list
    def size(self):
        return len(self.S)

    ## @brief Checks if both the Sets are equal
    #  @details Checks if the current set and the parameter set are both of the same length
    #           and then if elements in both the sets are equal
    #  @param R A Set ADT
    #  @return True if the length and the elements are the same, False otherwise
    def equals(self, r):
        if (len(r.S) == len(self.S)):
            for e in self.S:
                if not r.member(e):
                    return False
            return True
        return False

    ## @brief Returns a sequence of elements in the set
    #  @return List of elements in the set
    def to_seq(self):
        sequence = []
        for e in self.S:
            sequence.append(e)
        return sequence
```

```
## @brief Set index to start of the set
def __iter__(self):
    self.index = 0
    return self

## @brief Obtain value at current index and increase i value
#  @return Value at current index
#  @throw StopIteration Stops iterating when list out of range
def __next__(self):
    if self.size() > self.index:
        element = self.to_seq()[self.index]
        self.index = self.index + 1
        return element
    else:
        raise StopIteration
```

I Code for ElmSet.py

```
## @file ElemSet.py
#  @author Utsharga Rozario
#  @brief Module that creates a ElmSet ADT that inherits Set
#  @date Feb 08, 2020

from Set import *


## @brief An abstract data type that represents a sequence of elements
#  @details Inherits Set ADT and all it's methods
class ElmSet(Set):
    pass
```

J Code for MolecSet.py

```
## @file MolecSet.py
#  @author Utsharga Rozario
#  @brief Module that creates a MolecSet ADT that inherits Set
#  @date Feb 08, 2020

from Set import *


## @brief An abstract data type that represents a sequence of molecules of type MoleculeT
#  @details Inherits Set ADT and all it's methods
class MolecSet(Set):
    pass
```

K Code for CompoundT.py

```
## @file CompoundT.py
#  @author Utsharga Rozario
#  @brief Module that creates a CompoundT ADT
#  @date Feb 08, 2020

from Set import *
from ChemEntity import *
from ElmSet import *
from MolecSet import *
from MoleculeT import *


## @brief an Abstract Data Type that resemblems a compound
class CompoundT(ChemEntity, Equality):

    ## @brief Molecule Constructor
    #  @details Initializes a CompoundT object with an MolecSet of MoleculeT
    #  @param M Element in the molecule
    def __init__(self, m):
        self.C = m

    ## @brief Checks if both the compound are equal
    #  @details Checks if the current compound and the parameter compound have the same
    #             number of molecules and the same element of molecules
    #  @param other CompoundT ADT
    #  @return True if the Molecule and number of atoms of the element are the same,
    #            False otherwise
    def __eq__(self, other):
        if (self.C.equals(other.get_molec_set())):
            return True
        else:
            return False

    ## @brief Get the elements in compound and returns an ElmSet of it
    #  @return ElmSet of the elements in the compound
    def constit_elems(self):
        listelm = []
        for m in self.C.S:
            listelm.append(m.get_elm())
        return ElmSet(listelm)

    ## @brief Gets the molecule set of the compound
    #  @return The molecule set of the compound
    def get_molec_set(self):
        return self.C

    ## @brief Checks if both the compound are equal
    #  @details Checks if the current compound and the parameter compound have the same
    #             number of molecules and the same element of molecules
    #  @param other CompoundT ADT
    #  @return True if the Molecule and number of atoms of the element are the same,
    #            False otherwise
    def equals(self, d):
        if (self.C.equals(d.get_molec_set())):
            return True
        else:
            return False

    ## @brief Gets the number of atoms of the parameter element in the compound
    #  @param e ElementT type parameter
    #  @return The number atoms of the element in the Compound
    def num_atoms(self, e):
        atoms = 0
        for mol in self.C.S:
            atoms += mol.num_atoms(e)
        return atoms
```

L Code for ReactionT.py

```
## @file ReactionT.py
#  @author Utsharga Rozario
#  @brief Module that creates a ReactionT ADT
#  @date Feb 08, 2020

import numpy as np


## @brief an Abstract Data Type that resemblems a chemical reaction
class ReactionT:

    ## @brief Reaction Constructor
    #  @details Initializes a CompoundT object with an MolecSet of MoleculeT
    #  @param l A Sequence of Compounds Representing the left side of the reaction
    #  @param r A Sequence of Compounds Representing the right side of the reaction
    #  @throws ValueError if the coefficients of the compounds are negative or
    #          the reaction is not balanced by the produced coefficients
    def __init__(self, l, r):

        self.lhs = l
        self.rhs = r

        coefs = self.__cal_coef(self.lhs, self.rhs)

        self.CoefL = coefs[0]
        self.CoefR = coefs[1]

        if not (self.__pos(self.CoefL)):
            raise ValueError
        elif not (self.__pos(self.CoefR)):
            raise ValueError
        elif not (self.__is_balanced(self.lhs, self.rhs, self.CoefL, self.CoefR)):
            raise ValueError

    ## @brief Calculates the coefficients of the right and left side of the reaction
    #  Source: https://codegolf.stackexchange.com/questions/8728/balance-chemical-equations
    #
       https://www.nayuki.io/res/chemical-equation-balancer-javascript/chemical-equation-balancer.js
    #  @param lhs A Sequence of Compounds Representing the left side of the reaction
    #  @param rhs A Sequence of Compounds Representing the right side of the reaction
    #  @return A tuple containing the two lists, one for the left and one for the
    #          right hand side of the reaction, each containing the corresponding coefficients
    def __cal_coef(self, lhs, rhs):
        elmlistl = self.__make_seq(lhs)
        elmlistr = self.__make_seq(rhs)
        self.__is_list_valid(elmlistl, elmlistr)
        arr = self.__matrix(lhs, rhs, elmlistl)
        arr2 = self.__inv_matrix(arr)
        coeffs = self.__make_coeff(arr2, lhs, rhs)
        return coeffs

    ## @brief Makes a sequence of the elements in the Compound
    #  @param side A Sequence of Compounds Representing the one side of the reaction
    #  @return A sequence of elements from the Compound without any duplicates
    def __make_seq(self, side):
        elmlist = []
        elmlisttry = []
        for comp in side:
            elmlisttry = elmlisttry + comp.constit_elems().S
        for i in elmlisttry:
            if i not in elmlist:
                elmlist.append(i)
        return elmlist

    ## @brief Validates if the right side of the reaction has the same elements as the left
    #  @param elmlistl A sequence of elements representing the elements on the left side
    #  @param elmlistr A Sequence of elements representing the elements on the right side
    #  @throws ValueError If either the length of the right side elements' list and the
    #          left side elements' list do not match or one side contains a different element
    def __is_list_valid(self, elmlistl, elmlistr):
        if not (len(elmlistl) == len(elmlistr)):
            raise ValueError("Elements are different on both sides")
        for e in elmlistr:
            if not (e in elmlistr):
                raise ValueError("Elements mismatch on either of the sides")
```

```
## @brief Creates a square matrix representing the reaction and elements
#  @details Creates an identity matrix utilizing numpy, it then adds the corresponding
#           number of atoms of each element according to the elmlistl
#  @param lhs A Sequence of Compounds Representing the left side of the reaction
#  @param rhs A Sequence of Compounds Representing the right side of the reaction
#  @param elmlistl A sequence of elements from the left hand side without any duplicates
#  @return arr A square matrix representing the reaction and elements
def __matrix(self, lhs, rhs, elmlistl):
    cols = len(lhs) + len(rhs)
    rows = len(elmlistl)
    if rows > cols:
        arr = np.identity(rows, dtype=int)
    elif rows < cols:
        arr = np.identity(cols, dtype=int)
    for i in range(len(elmlistl)):
        j = 0
        for comp in lhs:
            arr[i][j] = comp.num_atoms(elmlistl[i])
            j += 1
        for comp in rhs:
            arr[i][j] = (-1) * comp.num_atoms(elmlistl[i])
            j += 1
    return arr

## @brief Inverses a matrix and returns the inversed matric in whole numbers
#  @details Uses numpy to inverse matrix and the fuction gcd to get whole numbers
#  @param arr A square matrix representing the reaction and elements
#  @return arr2 A matrix representing the inverse of the input matrix in whole numbers
def __inv_matrix(self, arr):
    arr2 = np.linalg.inv(arr)
    num1 = arr2[0][0]
    num2 = arr2[0][1]
    gcd = self.__find_gcd(num1, num2)
    for i in range(len(arr2)):
        for j in range(len(arr2[0])):
            gcd = self.__find_gcd(gcd, arr2[i][j])
    arr2 = (1 / gcd) * arr2
    return arr2

## @brief Creates the coefficients of the compounds of both sides of the reaction
#  @param lhs A Sequence of Compounds Representing the left side of the reaction
#  @param rhs A Sequence of Compounds Representing the right side of the reaction
#  @param arr A matrix representing the reaction coefficients which is the last
#         element of each row
#  @return arr A square matrix representing the reaction and elements
def __make_coeff(self, arr, lhs, rhs):
    cols = len(lhs) + len(rhs)
    newlist = []
    for i in range(len(arr)):
        newlist.append(arr[i][cols - 1])
    listleft = []
    listright = []
    for i in range(len(self.lhs)):
        listleft.append(int(newlist[i]))
    for i in range(len(listleft), len(newlist)):
        listright.append(int(newlist[i]))
    return (listleft, listright)

## @brief Finds the greatest common denominator of the two numbers
#  Source: https://www.geeksforgeeks.org/gcd-two-array-numbers/
#  @param x A number mostly expected to be rational
#  @param y A number mostly expected to be rational
#  @return x The greatest common denominator
def __find_gcd(self, x, y):
    while(y):
        x, y = y, x % y
    return x

## @brief Returns the sequence of compounds representing the left hand
#         side of the reaction
def get_lhs(self):
    return self.lhs

## @brief Returns the sequence of compounds representing the right hand
#         side of the reaction
def get_rhs(self):
    return self.rhs

## @brief Returns the sequence of coefficients representing the left hand
#         side coefficients of the reaction
```

```
def get_lhs_coeff(self):
    return self.CoefL

## @brief Returns the sequence of coefficients representing the right hand
#         side coefficients of the reaction
def get_rhs_coeff(self):
    return self.CoefR

## @brief Checks if the elements in the list are positive
#  @param coef A sequence of coefficients representing the coefficients of
#         one side of the reaction
#  @return True if all the elements are positive, False otherwise
def __pos(self, coef):
    for i in range(len(coef)):
        if coef[i] < 0:
            return False
    return True

## @brief Get the number of the atoms of one element in one side of the reaction
#  @param seq A sequence of compounds representing one side of the reaction
#  @param coef A sequence of compound coeffiecients representing one side of the reaction
#  @param e An ElementT type data
#  @return atoms The number of atoms of an element in one sode
def __n_atoms(self, seq, coef, e):
    atoms = 0
    for i in range(len(seq)):
        atoms = atoms + coef[i] * seq[i].num_atoms(e)
    return atoms

## @brief Gets and returns a sequence of elements in one side of the reaction
#  @param seq A sequence of compounds representing one side of the reaction
#  @return elmlist A sequence of elements in one side of the reaction
def __elm_in_chem_eq(self, seq):
    elmlist = []
    elmlisttry = []
    for comp in seq:
        elmlisttry = elmlisttry + comp.constit_elems().S
    for i in elmlisttry:
        if i not in elmlist:
            elmlist.append(i)
    return elmlist

## @brief Checks if an element is present with equal number of atoms on each side
#  @param l A sequence of compounds representing left side of the reaction
#  @param coefl A sequence of compound coeffiecients representing left side of the reaction
#  @param r A sequence of compounds representing right side of the reaction
#  @param coefr A sequence of compound coeffiecients representing right side of the reaction
#  @return True if an element is present with equal number of atoms on each side,
#          False otherwise
def __is_bal_elm(self, l, r, coefl, coefr, e):
    return self.__n_atoms(l, coefl, e) == self.__n_atoms(r, coefr, e)

## @brief Checks if the reaction is balanced
#  @param l A sequence of compounds representing left side of the reaction
#  @param coefl A sequence of compound coeffiecients representing left side of the reaction
#  @param r A sequence of compounds representing right side of the reaction
#  @param coefr A sequence of compound coeffiecients representing right side of the reaction
#  @return True if the reaction is balanced, False otherwise
def __is_balanced(self, l, r, coefl, coefr):
    self.__is_list_valid(self.__elm_in_chem_eq(l), self.__elm_in_chem_eq(r))
    for e in self.__elm_in_chem_eq(l):
        if not (self.__is_bal_elm(l, r, coefl, coefr, e)):
            return False
    return True
```

M Code for test_All.py

```
## @file test_All.py
#  @author Utsharga Rozario
#  @brief Tests implementation of python files Set, MoleculeT, CompoundT and ReactionT.
#  @date Feb 8, 2020
#  @details Written to test chemical reaction balancing
#           Avoids interacting with state variables to enforce information hiding.

import pytest

from ChemTypes import ElementT
from Set import *
from MoleculeT import *
from CompoundT import *
from ReactionT import *


## @brief Tests methods from Set.py
class TestSet:

    def setup_method(self, method):
        self.set1 = Set([1, 2, 3, 4, 5])
        self.set2 = Set([])

    def test_add(self):
        self.set1.add(6)
        assert self.set1.member(6)
        self.set1.add("32")
        assert self.set1.member("32")

    def test_rm(self):
        self.set1.rm(5)
        assert not self.set1.member(5)

        with pytest.raises(ValueError):
            self.set1.rm(7)

    def test_member(self):
        assert self.set1.member(2)
        assert not self.set1.member(8)

    def test_size(self):
        assert self.set1.size() == 5
        assert self.set2.size() == 0

    def test_equals(self):
        assert self.set1.equals(Set([1, 2, 3, 4, 5]))
        assert not self.set1.equals(Set([1, 2, 3, 4]))

    def test___eq__(self):
        assert self.set1 == Set([1, 2, 3, 4, 5])
        assert not self.set1 == Set([1, 2, 3, 4])

    def test_to_seq(self):
        assert self.set1.to_seq() == [1, 2, 3, 4, 5]


## @brief Tests methods from MoleculeT.py
class TestMoleculeT:

    def setup_method(self, method):
        self.molec1 = MoleculeT(5, ElementT.H)

    def test_get_elm(self):
        assert self.molec1.get_elm() == ElementT.H

    def test_get_num(self):
        assert self.molec1.get_num() == 5

    def test_equals(self):
        assert self.molec1.equals(MoleculeT(5, ElementT.H))
        assert not self.molec1.equals(MoleculeT(4, ElementT.H))
        assert not self.molec1.equals(MoleculeT(5, ElementT.O))

    def test___eq__(self):
        assert self.molec1 == MoleculeT(5, ElementT.H)
        assert not self.molec1 == MoleculeT(4, ElementT.H)
        assert not self.molec1 == MoleculeT(5, ElementT.O)

    def test_num_atoms(self):
        assert self.molec1.num_atoms(ElementT.H) == 5
        assert self.molec1.num_atoms(ElementT.O) == 0

    def test_constit_elms(self):
        assert self.molec1.constit_elems() == ElmSet([ElementT.H])


## @brief Tests methods from CompoundT.py
class TestCompoundT:

    def setup_method(self, method):
        self.m1 = MoleculeT(3, ElementT.C)
        self.m2 = MoleculeT(8, ElementT.H)
        self.m3 = MoleculeT(3, ElementT.O)
        self.m4 = MoleculeT(1, ElementT.Mg)
        self.m5 = MoleculeT(1, ElementT.Og)
        self.m6 = MoleculeT(17, ElementT.O)
        self.compound1 = CompoundT(MolecSet([self.m1, self.m2, self.m3]))
        self.compound2 = CompoundT(MolecSet([self.m4, self.m3, self.m5]))
        self.compound3 = CompoundT(MolecSet([self.m4]))
        self.compound4 = CompoundT(MolecSet([self.m3, self.m4, self.m6]))
        self.elmset1 = ElmSet([ElementT.C, ElementT.H, ElementT.O])
        self.elmset2 = ElmSet([ElementT.Mg, ElementT.O, ElementT.Og])
        self.elmset3 = ElmSet([ElementT.Mg])

    def test_get_molec_set(self):
        assert self.compound1.get_molec_set() == MolecSet([self.m1, self.m2, self.m3])
        assert self.compound2.get_molec_set() == MolecSet([self.m4, self.m3, self.m5])
        assert self.compound3.get_molec_set() == MolecSet([self.m4])

    def test___eq__(self):
        assert self.compound1 == CompoundT(MolecSet([self.m1, self.m2, self.m3]))
        assert self.compound2 == CompoundT(MolecSet([self.m4, self.m3, self.m5]))
        assert self.compound3 == CompoundT(MolecSet([self.m4]))
        assert not self.compound2 == CompoundT(MolecSet([self.m4, self.m3, self.m6]))

    def test_equals(self):
        assert self.compound1.equals(CompoundT(MolecSet([self.m1, self.m2, self.m3])))
        assert self.compound2.equals(CompoundT(MolecSet([self.m4, self.m3, self.m5])))
        assert self.compound3.equals(CompoundT(MolecSet([self.m4])))
        assert not self.compound2.equals(CompoundT(MolecSet([self.m4, self.m3, self.m6])))

    def test_num_atoms(self):
        assert self.compound1.num_atoms(ElementT.H) == 8
        assert self.compound2.num_atoms(ElementT.Ag) == 0
        assert self.compound4.num_atoms(ElementT.O) == 20

    def test_constit_elms(self):
        assert self.compound1.constit_elems() == self.elmset1
        assert self.compound2.constit_elems() == self.elmset2
        assert self.compound3.constit_elems() == self.elmset3


## @brief Tests methods from ReactionT.py
class TestReactionT:

    def setup_method(self, method):
        self.m1 = MoleculeT(1, ElementT.Na)
        self.m2 = MoleculeT(1, ElementT.Cl)
        self.m3 = MoleculeT(1, ElementT.S)
        self.m4 = MoleculeT(2, ElementT.O)
        self.m5 = MoleculeT(2, ElementT.H)
        self.m6 = MoleculeT(1, ElementT.O)
        self.m7 = MoleculeT(2, ElementT.Na)
        self.m10 = MoleculeT(4, ElementT.O)
        self.m11 = MoleculeT(1, ElementT.H)
        self.compound1 = CompoundT(MolecSet([self.m1, self.m2]))
        self.compound2 = CompoundT(MolecSet([self.m3, self.m4]))
        self.compound3 = CompoundT(MolecSet([self.m5, self.m6]))
        self.compound4 = CompoundT(MolecSet([self.m4]))
        self.compound5 = CompoundT(MolecSet([self.m7, self.m3, self.m10]))
        self.compound6 = CompoundT(MolecSet([self.m11, self.m2]))
        self.lhs = [self.compound1, self.compound2, self.compound3, self.compound4]
        self.rhs = [self.compound5, self.compound6]
        self.lhs2 = [CompoundT(MolecSet([self.m5])), CompoundT(MolecSet([self.m4]))]
        self.rhs2 = [CompoundT(MolecSet([self.m5, self.m6]))]
        self.list = [self.compound1, self.compound2, self.compound3, self.compound4]
        self.reaction = ReactionT(self.lhs, self.rhs)
        self.reaction2 = ReactionT(self.lhs2, self.rhs2)

    def test_get_lhs(self):
        assert self.reaction.get_lhs() == self.list
        assert self.reaction2.get_lhs() == self.lhs2

    def test_get_rhs(self):
        assert self.reaction.get_rhs() == [self.compound5, self.compound6]
        assert self.reaction2.get_rhs() == self.rhs2

    def test_get_lhs_coeff(self):
        assert self.reaction.get_lhs_coeff() == [4, 2, 2, 1]
        assert self.reaction2.get_lhs_coeff() == [2, 1]

    def test_get_rhs_coeff(self):
        assert self.reaction.get_rhs_coeff() == [2, 4]
        assert self.reaction2.get_rhs_coeff() == [2]
```

N Code for Partner's Set.py

```
## @file Set.py
#  @author Zihao Du
#  @brief Module that creates the Set generic abstract data type
#  @date Feb 6, 2020

from Equality import *


## @brief A generic abstract data type that represents a set
class Set(Equality):

    ## @brief Set constructor
    #  @details Initializes a Set object whose state consists
    #  of a sequence of some type
    #  @param s sequence of some type
    def __init__(self, s):
        self._S = set(s)

    ## @brief Add an element into the Set object
    def add(self, e):
        self._S.add(e)

    ## @brief Remove a certain element in the set
    #  @throw If the element is not in the set, a ValueError will be raised
    # @param e The element the client wants to delete
    def rm(self, e):
        if (self.member(e) is False):
            raise ValueError
        self._S.remove(e)

    ## @brief Determine if a certain element is in the set
    # @return True if the element is in the set, False otherwise
    # @param e The element the client want to test
    def member(self, e):
        for x in self._S:
            if (x == e):
                return True
        return False

    ## @brief Obtain the number of elements in the Set
    # @return The number of elements in the Set
    def size(self):
        return len(self._S)

    ## @brief Inherit from Equality,
    #  determine if a certain Set object equals the current one
    # @return True if they are the same set, False otherwise
    # @param r The Set object to compare with the current Set
    def equals(self, r):
        if (r.size() != self.size()):
            return False
        for e in self._S:
            if (r.member(e) is False):
                return False
        return True

    ## @brief Obtain the list of elements in the set
    # @return List of elements in the set
    def to_seq(self):
        return list(self._S)

    ## @brief Magic function, redefine the meaning of equivalence
    # @return True if two sets are the same set
    # @param r The Set object to compare with the current Set
    def __eq__(self, r):
        if (r.size() != self.size()):
            return False
        for e in self._S:
            if (r.member(e) is False):
                return False
        return True
```

O Code for Partner's MoleculeT.py

```python
## @file ChemTypes.py
#  @author Zihao Du
#  @brief Module that creates the MoleculeT ADT
#  @date Feb 6, 2020

from ChemTypes import *
from ElmSet import *
from ChemEntity import *
from Equality import *


## @brief An abstract data type that represents a Molecule, a child class of
#  ChemEntity and Equality
class MoleculeT(ChemEntity, Equality):

    ## @brief MoleculeT constructor
    #  @details Initializes a MoleculeT object whose state consists
    #  of a natural number and a ElementT object
    #  @param n a natural number(the subscript)
    #  @param e the ElementT object
    def __init__(self, n, e):
        self._num = n
        self._elm = e

    ## @brief Get number of atoms in the molecule
    # @return The number of atoms in the molecule
    def get_num(self):
        return self._num

    ## @brief Get element of the molecule
    # @return The element of the molecule
    def get_elm(self):
        return self._elm

    ## @brief Obtain the number of a certain element in the molecule
    # @return The number of atoms of the element in the molecule
    # @param e ElementT object the client is interested in
    def num_atoms(self, e):
        if (e != self._elm):
            return 0
        else:
            return self._num

    ## @brief Get the element in the molecule in a ElmSet
    # @return ElmSet that contains the element in the molecule
    def constit_elems(self):
        s1 = ElmSet([self._elm])
        return s1

    ## @brief Determine if a molecule is equal to the current molecule
    # @return True if they are the same, otherwise False
    # @param m A moleculeT object to compare with the current one
    def equals(self, m):
        if((m._elm == self._elm) & (m._num == self._num)):
            return True
        else:
            return False
```

P Code for Partner's CompoundT.py

```python
## @file CompoundT.py
#  @author Zihao Du
#  @brief Module that creates the CompoundT ADT
#  @date Feb 6, 2020

from MoleculeT import *
from MolecSet import *
from ElmSet import *
from ChemEntity import *
from Equality import *


## @brief An abstract data type that represents a compound(set of molecules), a
#  child class of ChemEntity and Equality
class CompoundT(ChemEntity, Equality):

    ## @brief CompoundT constructor
    #  @details Initializes a CompoundT object whose state consists
    #  of a MolecSet
    #  @param m A MolecSet
    def __init__(self, m):
        self.__C = m

    ## @brief Obtain the MolecSet of the compound
    # @return The MolecSet of the compound
    def get_molec_set(self):
        return self.__C

    ## @brief Inherit from ChemEntity,
    #  obtain the number of atoms of a certain element in the compound
    # @return The number of atoms of a certain element in the compound
    # @param e ElementT object the client is interested in
    def num_atoms(self, e):
        sum = 0
        for m in (self.__C).to_seq():
            sum += m.num_atoms(e)
        return sum

    ## @brief Inherit from ChemEntity,
    #  obtain the set of Elements in the compound
    # @return An ElmSet containing all elements that appears in the compound
    def constit_elems(self):
        set1 = ElmSet([])
        for m in (self.__C).to_seq():
            set1.add(m.get_elm())
        return set1

    ## @brief Inherit from Equality,
    #  determine if two compounds are the same
    # @return True if two compounds have the same molecules inside
    # @param d CompoundT object to compare with the current one
    def equals(self, d):
        return (self.__C.equals(d.get_molec_set()))

    ## @brief Magic method, determine if two compounds are the same
    # @return True if two compounds have the same molecules inside
    # @param d CompoundT object to compare with the current one
    def __eq__(self, d):
        return (self.__C.equals(d.get_molec_set()))
```

Q Code for Partner's ReactionT.py

```
## @file ReactionT.py
#  @author Zihao Du
#  @brief Module that creates the ReactionT ADT
#  @date Feb 6, 2020

from ChemTypes import *
from CompoundT import *
from numpy import *


## @brief An abstract data type that represents a chemical reaction
class ReactionT:

    ## @brief ReactionT constructor
    #  @details Initializes a ReactionT object whose state consists
    #  of two sequences of CompoundT, two sequences of real numbers.
    #  The constructor will calculate the two sequences of coefficients using
    #  linear system of equations solver.
    #  @throw If any coefficient is zero or negative,
    #  or the number of compounds is not number of elements plus one,
    #  or the reaction cannot be balanced, a ValueError will be raised.
    #  @param l sequence of CompoundT representing the left hand side of the reaction
    #  @param r sequence of CompoundT representing the right hand side of the reaction
    def __init__(self, l, r):
        if (__elm_in_chem_eq__(l) != __elm_in_chem_eq__(r)):
            raise ValueError
        elements = __elm_in_chem_eq__(l)
        a = []
        b = []
        for elm in elements.to_seq():
            b.append([-(l[0].num_atoms(elm))])
            row = []
            for m in (l + r)[1:]:
                row.append(m.num_atoms(elm))
            a.append(row)
        if (len(a) != len(a[0])):
            raise ValueError
        coeff = linalg.solve(a, b)
        lhsc = [1]
        rhsc = []
        for i in range(len(l) - 1):
            lhsc.append(float(coeff[i][0]))
        for i in range(len(r)):
            rhsc.append(float(coeff[len(l) - 1 + i][0]))
        rhsc = [-x for x in rhsc]
        if(__is_balanced__(l, r, lhsc, rhsc) is False):
            raise ValueError
        if (__pos__(rhsc) is False or __pos__(lhsc) is False):
            raise ValueError
        self._lhs = l
        self._rhs = r
        self._coeffl = lhsc
        self._coeffr = rhsc

    ## @brief Obtain the sequence of CompoundT representing the left hand side
    # @return The sequence of CompoundT representing the left hand side
    def get_lhs(self):
        return self._lhs

    ## @brief Obtain the sequence of CompoundT representing the right hand side
    # @return The sequence of CompoundT representing the right hand side
    def get_rhs(self):
        return self._rhs

    ## @brief Obtain the list of real numbers representing the left hand side coefficients
    # @return The sequence of real numbers representing the left hand side coefficients
    def get_lhs_coeff(self):
        return self._coeffl

    ## @brief Obtain the list of real numbers representing the right hand side coefficients
    # @return The sequence of real numbers representing the right hand side coefficients
    def get_rhs_coeff(self):
        return self._coeffr


## @brief Determine if a sequence of real numbers have a positive or zero element in it
```

```
#  @param s List of real numbers
#  @return True if all elements are positive, otherwise False
def __pos__(s):
    for m in s:
        if (m <= 0):
            return False
    return True


## @brief Count the number of a certain atom on one side of the reaction
#  @param cs List of CompoundT
#  @param c List of coefficients
#  @param e A certain Element the client wants to count
#  @return The number of atoms of a certain element in one side of the reaction
def __n_atoms__(cs, c, e):
    sum = 0
    for i in range(len(cs)):
        sum += (c[i] * cs[i].num_atoms(e))
    return sum


## @brief List all the elements that appear in a sequence of CompoundT
#  @param C List of CompoundT
#  @return A ElmSet of elements that appears in the sequence of compounds
def __elm_in_chem_eq__(c):
    s = ElmSet([])
    for com in c:
        for e in (com.constit_elems().to_seq()):
            s.add(e)
    return s


## @brief Determine if two sides of a reaction achieve a balance in a certain element
#  @param l List of CompoundT on one side
#  @param r List of CompoundT on the other side
#  @param cl List of coefficients on one side
#  @param cl List of coefficients on the other side
#  @param e A certain Element the client wants to count
#  @return True if the number of atoms of the element is equal, otherwise False
def __is_bal_elm__(l, r, cl, cr, e):
    return (__n_atoms__(l, cl, e) == __n_atoms__(r, cr, e))


## @brief Determine if two sides of a reaction is balanced
#  @param l List of CompoundT on one side
#  @param r List of CompoundT on the other side
#  @param cl List of coefficients on one side
#  @param cl List of coefficients on the other side
#  @return True if they achieve a balance, otherwise False
def __is_balanced__(l, r, cl, cr):
    for e in (__elm_in_chem_eq__(l)).to_seq():
        if (__is_bal_elm__(l, r, cl, cr, e) is False):
            return False
    if (__elm_in_chem_eq__(l) != __elm_in_chem_eq__(r)):
        return False
    return True
```