Assignment 2 Solution

Jay Mody

February 16, 2020

This report outlines the results of implementing a simplified chemical equation balancing program as specified by the provided MIS. The MIS includes a mix of interfaces and modules with generability and modularity in mind. This report will also discuss test results, critiques of the given design specification, and answer questions about software principles/design in general.

1 Assumptions

- To make testing simple, I assumed that the calculated reaction coefficients must be in their lowest whole number form.
- Since python is weakly typed langauge, some of the specifications weren't directly addressed. Technically, ElmSet and MolecSet are the same as just Set. For example, ElmSet is suppose to only contain elements of type ElementT as per the MIS, but there are actually no restrictions on the element types that may be added to an ElmSet.
- Any functions that are suppose to accept parameters of type ElmSet or MolecSet are required to have inputs of that type, otherwise a ValueError is raised (for example, CompoundT's input must be of type MolecSet).

2 Test Cases Rationale

Similar to assignment 1, my test cases covered everything from normal intended use, to boundary cases, to exceptions. Since there was an emphasis on formal specifications (MIS) for this assignment, I wanted to make sure I covered type testinig for the inputs for the module constructors. Also, I tested the immutability and encapsulation of objects to ensure the software was robust. For boolean returning unit tests, I made sure to cover both True and False cases (sometimes functions will always return True, so if you only had True cases, the bug would fly under your radar). I tried to fit 2-4 test cases per function, one for normal use, and the rest for trickier edge cases. Of all the modules however, I was the most rigirous with ReactionT, as that was the module with the most complex/error prone code in my opinion.

3 Results of Testing the Original Program

Here's a summary of the pytest results for testAll.py on my code:

Name	Stmts	Miss	Cover
src/A2Examples.py	34	34	0%
src/ChemEntity.py	5	2	60%
src/ChemTypes.py	120	0	100%
src/CompoundT.py	18	0	100%
src/ElmSet.py	3	0	100%
src/Equality.py	5	1	80%
src/MolecSet.py	3	0	100%
src/MoleculeT.py	25	0	100%
src/ReactionT.py	42	2	95%
src/Set.py	19	0	100%
src/test_All.py	208	0	100%
TOTAL	482	39	92%

========== 23 passed, in 0.39s ==========

All 23 unit tests were passed, with the coverage for all the modules at almost 100%. Testing revealed a couple of errors, mostly in ReactionT, but I was able to fix them all before the deadline.

4 Results of Testing Partner's Code

Here's a summary of the pytest results for testAll.py on my partner's code:

Name	Stmts	Miss	Cover
src/A2Examples.py	34	34	0%
src/ChemEntity.py	5	2	60%
src/ChemTypes.py	120	0	100%
src/CompoundT.py	19	0	100%
src/ElmSet.py	3	0	100%
src/Equality.py	5	2	60%
src/MolecSet.py	3	0	100%
src/MoleculeT.py	23	0	100%
src/ReactionT.py	62	6	90%
src/Set.py	25	0	100%
src/test_All.py	208	14	93%
TOTAL	507	58	89%

==== 3 failed, 20 passed in 0.95s ====

5 Critique of Given Design Specification

6 Answers

a) The main advantage of the formal specification over the natural langauge one from A1 was that there was a lot less ambiguity. Inputs, outputs, types, and similar properties were all strictly specified, leaving little room for intepretation. However, the huge disadvantage was that I had to to spend a good half of my time working on this project simply trying to understand what each module did, how they function in relation to one another, and the semantics of the program. All in all, the MIS specification was more frustrating to deal with as it was easier to make the interpretations in A1 than it was trying to understand the semantics of A2. Ideally, a design specification has both elements in my opinion. A detalied MIS as well as a natural language description of the underlying semantics.

b) The process of converting strings to logical syntatic components is called parsing, where the string is iterated such that it's substrings are interpreted to create internal object

representations of what that string is suppose to be (for example, for this assignment, the string "H2" would convert to MoleculeT(2, ElementT.H)). One way this can be implemented is to have each module have a parseString() method, where it accepts a string as input, and returns the corresponding object, or raises an error if the string is not a valid representation of the object. This way, there is no need to create any additional modules. ReactionT would use CompoundT's parseString, which would use MoleculeT's parseString, which would use ElementT's parseString. For example:

```
class ReactionT:
  def parse_string(string):
    try:
      lhs, rhs = string.split("=")
      lhs = [CompoundT.parse_string(s) for s in lhs.split()]
      rhs = [CompoundT.parse_string(s) for s in rhs.split()]
    except:
      raise ValueError("invalid string")

    return ReactionT(lhs, rhs)
```

c) Assuming that the atomic number of an element is equivalent to it's mass, with the current implementation, we can simply use the enumerated value of the element to calculate weight. You would add a function to calculate the mass in MoleculeT.py and CompoundT.py that would compute the following sum:

$(+e : \forall e | e \in ChemEntity.constit\_elems() : ChemEntity.num\_atoms(e) * e)$

However, since the atomic number isn't the best representation of the mass of an element, we can be more accurate by redesigning ElementT such that an element has a unique id (the atomic number) as well as the actual mass (). In this case, the sum might look like this:

$(+e : \forall e | e \in ChemEntity.constit\_elems() : ChemEntity.num\_atoms(e) * e.get\_mass())$

d) In chemistry, balanced equations must be written with the lowest whole number positive coefficients. This means that fractional coefficients are not allowed, as it doesn't make sense to have a third of a compound, with the excpetion of some molecules, like O_2, that are allowed to have $\frac{1}{2}$ as their coefficient. In general, *rational* real number coefficients aren't mathematically "wrong" since they're just scaled from the "right" answer by some scalar. However, if one or more of the coefficients is an irrational num-

ber, then there is no solution, but the way our software is designed, such a result would misinformatively spit out an answer. One (very inefficient but correct) way to calculate the whole right answer would be to simply multiply the coefficients by 1,2,3, etc... until you find a configuration in which all the coefficients are whole. Source: `https://www.nyu.edu/classes/tuckerman/adv.chem/lectures/lecture_2/node3.html`

e) Dynamic typing means that variables, functions, and parameters are given types during runtime, while static typing means that the types must be known/declared during compilation. One advantage of static typing is that it is more robust. For example, if we wanted to write a function to calculate if the substring "2AA4" was in a string, we might write in Python (which is dynamic):

```
def check2AA4(mystring):
    return "2AA4" in mystring
```

Unfortunately, this has the unintended consequence of also working with list (if the element "2AA4" exists in a list). This could lead to many headscratching bugs that occur during the runtime rather than during compile time. On the other hand, the advantage of dynamic programming is that writing code becomes easier, with less overhead and repetitive code.

f)
```
[(i, i+2) for i in range(1, 10-2, 2)]
```

g)
```
def my_funky_len_function(mylist):
    return sum(map(lambda e : 1, mylist))
```

h) An interface is ....., an implementation is .....

i) **Abstraction**:

Anticipation of change:

Generality:

Modularity:

Seperation of concerns:

G Code for ChemTypes.py

```
## @file ChemTypes.py
#  @author Jay Mody
#  @brief Provides module for chemistry types (ElementT).
#  @date 08/02/20 (dd/mm/yy)

from enum import Enum, auto

## @brief A module that represents the elements in the periodic table.
class ElementT(Enum):
    H = auto()
    He = auto()
    Li = auto()
    Be = auto()
    B = auto()
    C = auto()
    N = auto()
    O = auto()
    F = auto()
    Ne = auto()
    Na = auto()
    Mg = auto()
    Al = auto()
    Si = auto()
    P = auto()
    S = auto()
    Cl = auto()
    Ar = auto()
    K = auto()
    Ca = auto()
    Sc = auto()
    Ti = auto()
    V = auto()
    Cr = auto()
    Mn = auto()
    Fe = auto()
    Co = auto()
    Ni = auto()
    Cu = auto()
    Zn = auto()
    Ga = auto()
    Ge = auto()
    As = auto()
    Se = auto()
    Br = auto()
    Kr = auto()
    Rb = auto()
    Sr = auto()
    Y = auto()
    Zr = auto()
    Nb = auto()
    Mo = auto()
    Tc = auto()
    Ru = auto()
    Rh = auto()
    Pd = auto()
    Ag = auto()
    Cd = auto()
    In = auto()
    Sn = auto()
    Sb = auto()
    Te = auto()
    I = auto()
    Xe = auto()
    Cs = auto()
    Ba = auto()
    La = auto()
    Ce = auto()
    Pr = auto()
    Nd = auto()
    Pm = auto()
    Sm = auto()
    Eu = auto()
    Gd = auto()
    Tb = auto()
    Dy = auto()
    Ho = auto()
```

```
Er = auto()
Tm = auto()
Yb = auto()
Lu = auto()
Hf = auto()
Ta = auto()
W = auto()
Re = auto()
Os = auto()
Ir = auto()
Pt = auto()
Au = auto()
Hg = auto()
Tl = auto()
Pb = auto()
Bi = auto()
Po = auto()
At = auto()
Rn = auto()
Fr = auto()
Ra = auto()
Ac = auto()
Th = auto()
Pa = auto()
U = auto()
Np = auto()
Pu = auto()
Am = auto()
Cm = auto()
Bk = auto()
Cf = auto()
Es = auto()
Fm = auto()
Md = auto()
No = auto()
Lr = auto()
Rf = auto()
Db = auto()
Sg = auto()
Bh = auto()
Hs = auto()
Mt = auto()
Ds = auto()
Rg = auto()
Cn = auto()
Nh = auto()
Fl = auto()
Mc = auto()
Lv = auto()
Ts = auto()
Og = auto()
```

H Code for ChemEntity.py

```
## @file ChemEntity.py
#  @author Jay Mody
#  @brief Provides module interface for chemical entities (molecules, compounds, etc ...).
#  @date 08/02/20 (dd/mm/yy)

## @brief A module interface for chemical entities.
#  @details A module interface for chemical entities like molecules and compounds.
class ChemEntity:

    ## @brief Counts the number of atoms of a specific element.
    #  @param e The ElementT atom to count.
    #  @return An integer for the number of atoms e.
    def num_atoms(self, e):
        raise NotImplementedError

    ## @brief Gets the set of all elements that constitute this entity.
    #  @return An ElmSet of all the ElementT's in this entity.
    def constit_elems(self):
        raise NotImplementedError
```

I Code for Equality.py

```
## @file Equality.py
#  @author Jay Mody
#  @brief Provides generic interface module for equality.
#  @date 08/02/20 (dd/mm/yy)

## @brief A generic interface for modules with equality properties.
class Equality:
    ## @brief Determines if this object is equivalent to object T.
    #  @param T The object to be compared with.
    #  @returns A boolean that is True if T is equivalent to this object, else False
    def equals(self, T):
        raise NotImplementedError

    ## @brief Allows equivalency comparisons via ==
    #  @details If A and B are objects that implement Equality, then this function makes A == B the
        same as A.equals(B).
    #  @returns A boolean that is True if T is equivalent to this object, else False
    def __eq__(self, T):
        return self.equals(T)
```

J Code for Set.py

```
## @file Set.py
#  @author Jay Mody
#  @brief Provides generic template module for Sets.
#  @date 08/02/20 (dd/mm/yy)

from Equality import Equality

## @brief A generic module for Sets.
class Set(Equality):

    ## @brief Constructor for Set objects.
    #  @return A list of input elements.
    def __init__(self, s):
        self.__set = set(s)

    ## @brief Adds an element to the set.
    #  @param e The element to add.
    def add(self, e):
        self.__set.add(e)

    ## @brief Removes an element from the set.
    #  @param e The element to remove.
    #  @throws ValueError Thrown if e is not in the set.
    def rm(self, e):
        try:
            self.__set.remove(e)
        except KeyError:
            raise ValueError("Element {} is not in set.".format(e))

    ## @brief Checks if an element exists in the set.
    #  @param e The element to check if it is a member of the set.
    #  @returns A boolean that is True if e is a member of set, else False
    def member(self, e):
        return e in self.__set

    ## @brief Gets the size of the set.
    #  @returns An integer for the number of elements in the set (size).
    def size(self):
        return len(self.__set)

    ## @brief Checks if this set is equivalent to set R.
    #  @param R The set to compare if this set is equal to it.
    def equals(self, R):
        return self.__set == R.__set and self.size() == R.size()

    ## @brief Returns a sequence of the set.
    #  @details The order of the elements in the sequence are not guarenteed to be consitent as a set
      is unordered.
    #  @returns A list of the elements in set.
    def to_seq(self):
        return list(self.__set)
```

K Code for ElmSet.py

```
## @file ElemSet.py
#  @author Jay Mody
#  @brief Provides Set module for ElementT objects.
#  @date 08/02/20 (dd/mm/yy)

from Set import Set

## @brief A module for Sets of elements with type ElementT.
class ElmSet(Set):
    pass
```

L Code for MolecSet.py

```
## @file MolecSet.py
#  @author Jay Mody
#  @brief Provides Set module for MoleculeT objects.
#  @date 08/02/20 (dd/mm/yy)

from Set import Set

## @brief A module for Sets of elements with type MoleculeT.
class MolecSet(Set):
    pass
```

M Code for CompoundT.py

```
## @file CompoundT.py
#  @author Jay Mody
#  @brief Provides module for chemical compounds.
#  @date 08/02/20 (dd/mm/yy)

from ChemEntity import ChemEntity
from Equality import Equality
from ElmSet import ElmSet
from MolecSet import MolecSet

import copy

## @brief A module for chemical compounds.
#  @details A module for chemical compounds, defined by a MolecSet.
class CompoundT(ChemEntity, Equality):

    ## @brief Constructor for CompoundT objects.
    #  @param m A MolecSet that defines the compound.
    #  @throws ValueError Thrown if m is not of type MolecSet.
    def __init__(self, m):
        if not isinstance(m, MolecSet):
            raise ValueError("m must be of type MolecSet")
        self.__c = copy.deepcopy(m)

    ## @brief Gets the MolecSet of this compound.
    #  @return m The MolecSet object for this compound.
    def get_molec_set(self):
        return self.__c

    ## @brief Gets the number of atoms of a given type in this molecule.
    #  @param e An ElementT atom to count for in the compound.
    #  @returns The number of atoms of type e in this compound.
    def num_atoms(self, e):
        return sum([m.num_atoms(e) for m in self.get_molec_set().to_seq()])

    ## @brief Gets the set of all elements that constitute this molecule.
    #  @return An ElmSet of all the ElementT's in this entity.
    def constit_elems(self):
        return ElmSet([m.get_elm() for m in self.get_molec_set().to_seq()])

    ## @brief Determines if this compound is equivalent to a compound D.
    #  @details A compound is considered equivalent to another if their MolecSets are equal.
    #  @param The CompoundT object to compare to this one.
    #  @return A boolean that is True if D is equivalent to this compound, else False.
    def equals(self, D):
        return self.get_molec_set().equals(D.get_molec_set())
```

N Code for ReactionT.py

```
## @file ReactionT.py
#  @author Jay Mody
#  @brief Provides module for chemical reactions.
#  @date 08/02/20 (dd/mm/yy)

import copy
import math
import collections
import numpy as np

## @brief A module to represent and balance chemical reactions.
class ReactionT:

    ## @brief Constructor for a ReactionT object.
    #  @details Constructs a chemical reaction and computes the correct coefficients for the reaction
      after being balanced by solving a linear system of equations.
    #  @param l A list of CompoundT objects that make up the left hand side of the equation.
    #  @param r A list of CompoundT objects that make up the right hand side of the equation.
    #  @throws ValueError Thrown if there are an infinite number of linearly independent solutions.
    #  @cite https://www.nyu.edu/classes/tuckerman/adv.chem/lectures/lecture_2/node3.html
    #  @cite https://stackabuse.com/solving-systems-of-linear-equations-with-pythons-numpy/
    def __init__(self, l, r):
        self.__lhs = copy.deepcopy(l)
        self.__rhs = copy.deepcopy(r)

        # find matrix of elements x compounds
        cols = len(l)+len(r)
        count = collections.defaultdict(lambda: [0]*cols)
        for i, compound in enumerate(l+r):
            for molec in compound.get_molec_set().to_seq():
                count[molec.get_elm()][i] = molec.get_num()

        # check if solution is unique
        rows = len(count)
        if cols-1 > rows:
            raise ValueError("There exists multiple independent solutions for reaction.")

        # linear algebra solve for coefficients
        mat = np.array(list(count.values()))
        lmat = mat[:,:len(l)]
        rmat = -1 * mat[:,len(l):-1]
        x = np.hstack((lmat, rmat))
        y =  mat[:,-1]
        soln = np.linalg.lstsq(x, y)[0]

        # convert coefficients to whole numbers
        soln = np.append(soln, 1)
        for i in range(1,1000):
            whole_soln = i*soln
            if all(abs(round(n) - n) < 1e-8 for n in whole_soln):
                soln = [round(n) for n in whole_soln]
                break

        self.__lhs_coeff = list(soln[:len(l)])
        self.__rhs_coeff = list(soln[len(l):])

        if not all(n > 0 for n in self.__lhs_coeff):
            raise ValueError("Invalid solution (negative/zero coeffs) found.")
        if not all(n > 0 for n in self.__rhs_coeff):
            raise ValueError("Invalid solution (negative/zero coeffs) found.")

    ## @brief Gets the left hand side of the equation.
    #  @return m A list of CompoundT objects that represent the left hand side of the equation.
    def get_lhs(self):
        return self.__lhs

    ## @brief Gets the right hand side of the equation.
    #  @return m A list of CompoundT objects that represent the right hand side of the equation.
    def get_rhs(self):
        return self.__rhs

    ## @brief Gets the coefficients for each compound on the left hand side of the equation.
    #  @details The compounds and coefficients are index related, so self.get_lhs_coeff()[i] would be
      the coefficient for compound self.get_lhs()[i].
    #  @return m A list of integers that represent the coefficients for the left hand side of the
      equation.
```

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

## @brief Gets the coefficients for each compound on the right hand side of the equation.
#   @details The compounds and coefficients are index related, so self.get_rhs_coeff()[i] would be
     the coefficient for compound self.get_rhs()[i].
#   @return m A list of integers that represent the coefficients for the right hand side of the
     equation.
def get_rhs_coeff(self):
    return self.__rhs_coeff
```

O Code for test_All.py

```
## @file test_All.py
#  @author Jay Mody
#  @brief Test driver for Set, MoleculeT, CompoundT, and ReactionT.
#  @date 08/02/20 (dd/mm/yy)

from Set import Set
from MoleculeT import MoleculeT
from CompoundT import CompoundT
from ReactionT import ReactionT
from ChemTypes import ElementT
from MolecSet import MolecSet
from ElmSet import ElmSet

import pytest
import numpy as np

# Set tests
def test_Set_init():
    assert Set([0, 1, 2])
    assert Set((0, "0", 1.2))

def test_Set_add():
    s = Set([1, 2, 3])

    s.add(12)
    assert s == Set([1, 2, 3, 12])

    s.add(3)
    assert s == Set([1, 2, 3, 12])

    s = Set([])

    s.add(1)
    assert s == Set([1])

    s.add("a")
    s.add("b")
    s.add("c")
    assert s == Set(["a", "b", "c", 1])

def test_Set_rm():
    with pytest.raises(ValueError):
        Set([]).rm(None)
    with pytest.raises(ValueError):
        Set([1]).rm(2)

    s = Set([1, 2, 3])
    s.rm(2)
    assert s == Set([1, 3])

    s = Set([1, 2, 3])
    s.rm(2)
    s.rm(1)
    s.rm(3)
    assert s == Set([])

def test_Set_member():
    assert not Set([]).member(True)
    assert not Set([1, 12, 2031]).member(-1)
    assert not Set([0.00001]).member(0)

    assert Set([1, 1, 1]).member(1)
    assert Set([1, 2, 3]).member(3)
    assert Set(["abc", 0]).member("abc")

def test_Set_size():
    assert Set([]).size() == 0
    assert Set([99]).size() == 1
    assert Set([1, 1, 1]).size() == 1
    assert Set(list(range(121))).size() == 121

def test_Set_equals():
    assert not Set([1, 2, 2012]).equals(Set([1, 2012]))
    assert not Set(["apple"]).equals(Set([]))

    assert Set([1, 2, 2012]).equals(Set([1, 2, 2012]))
```

```
    assert Set([1, 1, 1]).equals(Set([1]))


def test_Set_to_seq():
    assert isinstance(Set([1,2,3]).to_seq(), list)

    s = Set(list(range(2,20,3)))
    assert len(s.to_seq()) == s.size()


# MoleculeT tests
def test_MoleculeT_init():
    with pytest.raises(ValueError):
        MoleculeT(1, "He")
    with pytest.raises(ValueError):
        MoleculeT(0, ElementT.H)
    with pytest.raises(ValueError):
        MoleculeT(-1, ElementT.H)
    with pytest.raises(ValueError):
        MoleculeT(2.2, ElementT.He)

    assert MoleculeT(2, ElementT.H)
    assert MoleculeT(19999, ElementT.C)
    assert MoleculeT(1, ElementT.Fe)

def test_MoleculeT_get_num():
    assert MoleculeT(2, ElementT.O).get_num() == 2
    assert MoleculeT(1, ElementT.O).get_num() != 0

def test_MoleculeT_get_elm():
    assert MoleculeT(20, ElementT.O).get_elm() == ElementT.O
    assert MoleculeT(20, ElementT.H).get_elm() != 1

def test_MoleculeT_num_atoms():
    assert MoleculeT(20, ElementT.O).num_atoms(ElementT.O) == 20
    assert MoleculeT(10, ElementT.H).num_atoms(ElementT.O) == 0

def test_MoleculeT_constit_elems():
    assert MoleculeT(2, ElementT.H).constit_elems() == MoleculeT(1, ElementT.H).constit_elems()
    assert MoleculeT(3, ElementT.Mg).constit_elems() == ElmSet([ElementT.Mg])

def test_MoleculeT_equals():
    assert MoleculeT(2, ElementT.H).equals(MoleculeT(2, ElementT.H))
    assert MoleculeT(19999, ElementT.C) == MoleculeT(19999, ElementT.C)


# CompoundT tests
def test_CompoundT_init():
    m1 = MoleculeT(2, ElementT.H)
    m2 = MoleculeT(1, ElementT.O)

    with pytest.raises(ValueError):
        CompoundT([m1, m2])
    with pytest.raises(ValueError):
        CompoundT(set([m1, m2]))

    s = MolecSet([m1, m2])
    assert CompoundT(s)
    assert CompoundT(MolecSet([]))

def test_CompoundT_get_molec_set():
    m1 = MoleculeT(2, ElementT.H)
    m2 = MoleculeT(1, ElementT.O)
    s1 = MolecSet([m1, m2])

    m3 = MoleculeT(2, ElementT.H)
    m4 = MoleculeT(1, ElementT.O)
    s2 = MolecSet([m3, m4])

    c = CompoundT(s1)
    assert c.get_molec_set() == s2

    s1.add(MoleculeT(1, ElementT.He))
    print(s1.size())
    print(c.get_molec_set().size())
    assert c.get_molec_set() != s1

def test_CompoundT_num_atoms():
    m1 = MoleculeT(22, ElementT.H)
    m2 = MoleculeT(1, ElementT.O)
```

```
    m3 = MoleculeT(1, ElementT.O)
    c = CompoundT(MolecSet([m1, m2, m3]))
    assert c.num_atoms(ElementT.H) == 22
    assert c.num_atoms(ElementT.O) == 1
    assert c.num_atoms(ElementT.He) == 0

    m1 = MoleculeT(1, ElementT.Na)
    m2 = MoleculeT(1, ElementT.Cl)
    c = CompoundT(MolecSet([m1, m2]))
    assert c.num_atoms(ElementT.Na) == 1
    assert c.num_atoms(ElementT.Cl) == 1
    assert c.num_atoms(ElementT.H) == 0

def test_CompoundT_constit_elems():
    m1 = MoleculeT(2, ElementT.H)
    m2 = MoleculeT(1, ElementT.O)
    c = CompoundT(MolecSet([m1, m2]))
    assert c.constit_elems() == ElmSet([ElementT.H, ElementT.O])

    m1 = MoleculeT(1, ElementT.Na)
    m2 = MoleculeT(1, ElementT.Cl)
    c = CompoundT(MolecSet([m1, m2]))
    assert c.constit_elems() != ElmSet([ElementT.Na, ElementT.C])

def test_CompoundT_equals():
    m1 = MoleculeT(2, ElementT.H)
    m2 = MoleculeT(1, ElementT.O)
    s1 = MolecSet([m2, m1])

    m3 = MoleculeT(2, ElementT.H)
    m4 = MoleculeT(1, ElementT.O)
    s2 = MolecSet([m3, m4])

    assert CompoundT(s1) == CompoundT(s2)

    s2.add(MoleculeT(42, ElementT.U))
    assert CompoundT(s1) != CompoundT(s2)


# ReactionT tests
## @cite https://www.nayuki.io/page/chemical-equation-balancer-javascript
def test_ReactionT_init():
    # HClFe -> NO, is an invalid chemical equation
    m1 = MoleculeT(1, ElementT.H)
    m2 = MoleculeT(1, ElementT.Cl)
    m3 = MoleculeT(1, ElementT.Fe)
    m4 = MoleculeT(1, ElementT.N)
    m5 = MoleculeT(1, ElementT.O)
    lhs = [CompoundT(MolecSet([m1, m2, m3]))]
    rhs = [CompoundT(MolecSet([m4, m5]))]

    with pytest.raises(Exception):
        ReactionT(lhs, rhs)

    # H2 + O2 -> H2O, is a valid chemical equation that can be balanced
    m1 = MoleculeT(2, ElementT.H)
    m2 = MoleculeT(2, ElementT.O)
    m3 = MoleculeT(1, ElementT.O)
    lhs = [
        CompoundT(MolecSet([m1])),
        CompoundT(MolecSet([m2])),
    ]
    rhs = [
        CompoundT(MolecSet([m1, m3]))
    ]

    assert ReactionT(lhs, rhs)

def test_ReactionT_get_lhs():
    # H2 + O2 -> H2O
    m1 = MoleculeT(2, ElementT.H)
    m2 = MoleculeT(2, ElementT.O)
    m3 = MoleculeT(1, ElementT.O)
    lhs = [
        CompoundT(MolecSet([m1])),
        CompoundT(MolecSet([m2])),
    ]
    rhs = [
        CompoundT(MolecSet([m1, m3]))
    ]
```

```
    assert ReactionT(lhs, rhs).get_lhs() == lhs

def test_ReactionT_get_rhs():
    # H2 + O2 -> H2O
    m1 = MoleculeT(2, ElementT.H)
    m2 = MoleculeT(2, ElementT.O)
    m3 = MoleculeT(1, ElementT.O)
    lhs = [
        CompoundT(MolecSet([m1])),
        CompoundT(MolecSet([m2])),
    ]
    rhs = [
        CompoundT(MolecSet([m1, m3]))
    ]

    assert ReactionT(lhs, rhs).get_rhs() == rhs

def test_ReactionT_get_lhs_coeff():
    # H2 + O2 -> H20, coeffs [2, 1, 2]
    h2 = MoleculeT(2, ElementT.H)
    o2 = MoleculeT(2, ElementT.O)
    o1 = MoleculeT(1, ElementT.O)
    lhs = [
        CompoundT(MolecSet([h2])),
        CompoundT(MolecSet([o2])),
    ]
    rhs = [
        CompoundT(MolecSet([h2, o1]))
    ]
    assert ReactionT(lhs, rhs).get_lhs_coeff() == [2, 1]

    # Mg(OH)2 -> MgO + H20, coeffs [1, 1, 1]
    mg1 = MoleculeT(1, ElementT.Mg)
    lhs = [
        CompoundT(MolecSet([mg1, h2, o2])),
    ]
    rhs = [
        CompoundT(MolecSet([mg1, o1])),
        CompoundT(MolecSet([h2, o1]))
    ]
    assert ReactionT(lhs, rhs).get_lhs_coeff() == [1]

    # Mg(OH)2 -> MgO + H20, coeffs [2, 3, 4, 3]
    fe2 = MoleculeT(2, ElementT.Fe)
    fe1 = MoleculeT(1, ElementT.Fe)
    o3 = MoleculeT(3, ElementT.O)
    c1 = MoleculeT(1, ElementT.C)
    lhs = [
        CompoundT(MolecSet([fe2, o3])),
        CompoundT(MolecSet([c1])),
    ]
    rhs = [
        CompoundT(MolecSet([fe1])),
        CompoundT(MolecSet([c1, o2]))
    ]
    assert ReactionT(lhs, rhs).get_lhs_coeff() == [2, 3]

def test_ReactionT_get_rhs_coeff():
    # H2 + O2 -> H20, coeffs [2, 1, 2]
    h2 = MoleculeT(2, ElementT.H)
    o2 = MoleculeT(2, ElementT.O)
    o1 = MoleculeT(1, ElementT.O)
    lhs = [
        CompoundT(MolecSet([h2])),
        CompoundT(MolecSet([o2])),
    ]
    rhs = [
        CompoundT(MolecSet([h2, o1]))
    ]
    assert ReactionT(lhs, rhs).get_rhs_coeff() == [2]

    # Mg(OH)2 -> MgO + H20, coeffs [1, 1, 1]
    mg1 = MoleculeT(1, ElementT.Mg)
    lhs = [
        CompoundT(MolecSet([mg1, h2, o2])),
    ]
    rhs = [
        CompoundT(MolecSet([mg1, o1])),
        CompoundT(MolecSet([h2, o1]))
```

```
]
assert ReactionT(lhs, rhs).get_rhs_coeff() == [1, 1]

# Mg(OH)2 -> MgO + H20, coeffs [2, 3, 4, 3]
fe2 = MoleculeT(2, ElementT.Fe)
fe1 = MoleculeT(1, ElementT.Fe)
o3 = MoleculeT(3, ElementT.O)
c1 = MoleculeT(1, ElementT.C)
lhs = [
    CompoundT(MolecSet([fe2, o3])),
    CompoundT(MolecSet([c1])),
]
rhs = [
    CompoundT(MolecSet([fe1])),
    CompoundT(MolecSet([c1, o2]))
]
assert ReactionT(lhs, rhs).get_rhs_coeff() == [4, 3]
```

P Code for Partner's Set.py

```
## @file Set.py
#  @author Benjamin Kostiuk
#  @brief Module that defines the Set ADT
#  @details Assumes that the Set constructor is called for each object instance
#           before any other access methods are called.
#  @date 02/01/2020

from Equality import Equality


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

    ## @brief Set constructor
    #  @details Initializes a Set object whose state consists of a set of elements
    #  @param s Sequence of elements with which to initialize the Set
    def __init__(self, s):
        self.S = set(s)

    ## @brief Add an element to the set
    #  @param Element to be added to the set
    def add(self, e):
        self.S = self.S.union({e})

    ## @brief Remove an element from the set
    #  @param e Element to be removed from the set
    #  @throws ValueError if element to be removed cannot be found in the set
    def rm(self, e):
        if not self.member(e):
            raise ValueError("Cannot remove element not found in set.")
        self.S = self.S.difference({e})

    ## @brief Determine whether an element is in the set
    #  @param e Element to check whether in the set
    #  @return True if the element is in the set, otherwise false
    def member(self, e):
        return e in self.S

    ## @brief Get the size of the set
    #  @return The size of the set
    def size(self):
        return len(self.S)

    ## @brief Determine if the Set is equal to another set
    #  @details A Set is considered equal if all elements in one set are in another
    #  @param r Set to compare with
    #  @return True if the two Sets are equal, otherwise false
    def equals(self, r):
        if self.size() != r.size():
            return False

        for element in self.S:
            if not r.member(element):
                return False
        return True

    ## @brief Returns a sequence of all elements in the set
    #  @return A sequence of all elements in the set
    def to_seq(self):
        return list(self.S)

    def __eq__(self, value):
        return self.equals(value)
```

Q Code for Partner's MoleculeT.py

```
## @file MoleculeT.py
#  @author Benjamin Kostiuk
#  @brief Module defines the MoleculeT ADT for molecule representation
#  @date 02/01/2020

from Equality import Equality
from ChemEntity import *


## @brief An abstract data type that represents a molecule
class MoleculeT(ChemEntity, Equality):

    ## @brief MoleculeT constructor
    #  @details Initializes a MoleculeT object whose state consists of
    #           an ElementT and the number of that element in the molecule
    #  @param m Number of the ElementT in molecule
    #  @param e ElementT in the molecule
    def __init__(self, n, e):
        self.num = n
        self.elm = e

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

    ## @brief Get the ElementT in the molecule
    #  @return The ElementT in the molecule
    def get_elm(self):
        return self.elm

    ## @brief Get the number of atoms of a given ElementT in the molecule
    #  @param e ElementT to check for in molecule
    #  @return The number of atoms of the specified ElementT in the molecule
    def num_atoms(self, e):
        if self.elm == e:
            return self.num
        return 0

    ## @brief Return an ElmSet of the ElementT in the molecule
    #  @return An ElmSet of the ElementT in the molecule
    def constit_elems(self):
        return ElmSet([self.elm])

    ## @brief Determine if the molecule is equal to another molecule
    #  @details Two molecules are considered equal if they are composed of the same
    #           ElementT and have the same number of atoms.
    #  @param m MoleculeT to compare with
    #  @return True if the molecules are equal, otherwise false
    def equals(self, m):
        return self.elm == m.get_elm() and self.num == m.get_num()

    def __eq__(self, value):
        return self.equals(value)

    def __hash__(self):
        return hash(str(self.num) + str(self.elm))
```

R Code for Partner's CompoundT.py

```
## @file CompoundT.py
#  @author Benjamin Kostiuk
#  @brief Module defines the CompoundT ADT for chemical compound representation
#  @date 02/01/2020

from MoleculeT import *
from MolecSet import *


## @brief An abstract data type that represents a chemical compound
class CompoundT(ChemEntity, Equality):

    ## @brief CompoundT constructor
    #  @details Initializes a CompoundT object whose state consists of a MolecSet
    #  @param m MolecSet of molecules in the chemical compound
    def __init__(self, m):
        self.C = m

    ## @brief Get the MolecSet of molecules in the chemical compound
    #  @return The MolecSet of molecules in the chemical compound
    def get_molec_set(self):
        return self.C

    ## @brief Get the number of atoms of a given ElementT in the chemical compound
    #  @param e ElementT to check for in chemical compound
    #  @return The number of atoms of the specified ElementT in the chemical compound
    def num_atoms(self, e):
        count = 0
        for m in self.C.to_seq():
            count += m.num_atoms(e)
        return count

    ## @brief Return an ElmSet of the ElementTs in the chemical compound
    #  @return An ElmSet of the ElementTs in the chemical compound
    def constit_elems(self):
        return ElmSet([m.get_elm() for m in self.C.to_seq()])

    ## @brief Determine if the chemical compound is equal to another chemical compound
    #  @details Two chemical compounds are considered equal if they have
    #           all the same molecules in them
    #  @param d CompoundT to compare with
    #  @return True if the chemical compounds are equal, otherwise false
    def equals(self, d):
        return self.C.equals(d.get_molec_set())

    def __eq__(self, value):
        return self.equals(value)
```

S Code for Partner's ReactionT.py

```
## @file ReactionT.py
#  @author Benjamin Kostiuk
#  @brief Module defines the ReactionT ADT for representing checmical reactions
#  @date 02/05/2020

from CompoundT import *

import numpy as np
from sympy import Matrix, lcm


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

    ## @brief ReactionT constructor
    #  @details Initializes a ReactionT object whose state consists of a sequence of
    #           reactants a sequence of its coefficients, a sequence of products
    #           and a sequence of its coefficients. The sequences of coefficients
    #           are computed as to balance the chemical reaction with an equal
    #           number of elements on both sides.
    #  @param reactants Sequence of CompoundT in the left-hand side of the chemical
    #                   reaction, known as reactants
    #  @param products Sequence of CompoundT in the right-hand side of the chemical
    #                  reaction, known as products
    #  @throws ValueError if the elements in the reactants do not match the elements
    #                     in the products, the two sides of the reaction cannot be
    #                     balanced, any of coefficients are non-positive or if the
    #                     the sequences of coefficients do not match their
    #                     respective side of the chemical reaction
    def __init__(self, reactants, products):
        # Get ElmSet of ElementTs in L and R
        lhs_elems = self.__elements_in_equation__(reactants)
        rhs_elems = self.__elements_in_equation__(products)

        # Check that elements in the reactants and the products are the same
        if not lhs_elems.equals(rhs_elems):
            raise ValueError("Elements in reactants must match elements in products.")

        # Get coefficient matrix to solve linear equation
        lhs_coeffs, rhs_coeffs = self.__solve_matrix__(reactants, products, lhs_elems)

        # Check if length of lists match
        if len(lhs_coeffs) != len(reactants) or len(rhs_coeffs) != len(products):
            raise ValueError("Cannot match coefficients to reactants and products.")

        # Check if coefficients are balanced
        for element in lhs_elems.to_seq():
            if not self.__is_balanced__(reactants, products, lhs_coeffs, rhs_coeffs, element):
                raise ValueError("Invalid ReactionT. Reaction cannot be balanced.")

        self.lhs = reactants
        self.rhs = products
        self.coeff_L = lhs_coeffs
        self.coeff_R = rhs_coeffs

    ## @brief Get the sequence of reactants of the chemical reaction
    #  @return The sequence of CompoundT in the left-hand side of the chemical reaction
    def get_lhs(self):
        return self.lhs

    ## @brief Get the sequence of products of the chemical reaction
    #  @return The sequence of CompoundT in the right-hand side of the chemical reaction
    def get_rhs(self):
        return self.rhs

    ## @brief Get the sequence of coefficients in the left-hand side of the chemical reaction
    #  @return The sequence of coefficients in the left-hand side of the chemical reaction
    def get_lhs_coeff(self):
        return self.coeff_L

    ## @brief Get the sequence of coefficients in the right-hand side of the chemical reaction
    #  @return The sequence of coefficients in the right-hand side of the chemical reaction
    def get_rhs_coeff(self):
        return self.coeff_R

    # Returns an ElmSet of ElementT in a list of CompoundTs
```

```
def __elements_in_equation__(self, equation):
    elems = []
    for compound in equation:
        elems += compound.constit_elems().to_seq()
    return ElmSet(elems)

# Check if a ReactionTs coefficients for reactants and products are balanced
def __is_balanced__(self, reactants, products, left_coeffs, right_coeffs, element):
    lhs_count, rhs_count = 0, 0
    # Count element for left hand side
    for i in range(len(reactants)):
        if left_coeffs[i] <= 0:
            raise ValueError("Invalid ReactionT. Coefficients must be positive.")
        lhs_count += left_coeffs[i] * reactants[i].num_atoms(element)

    # Count element for right hand side
    for i in range(len(products)):
        if right_coeffs[i] <= 0:
            raise ValueError("Invalid reaction. Coefficients must be positive.")
        rhs_count += right_coeffs[i] * products[i].num_atoms(element)

    return lhs_count == rhs_count

# Return right and left coefficients solved from a list of reactants and products
def __solve_matrix__(self, reactants, products, elems):
    # Create a coefficient matrix to solve
    coeff_matrix = []
    for e in elems.to_seq():
        row = [compnd.num_atoms(e) for compnd in reactants]
        row += [-compnd.num_atoms(e) for compnd in products]
        coeff_matrix.append(row)

    # Check if reaction is null
    if(reactants == [] and products == []):
        return [], []
    else:
        # Solve for lhs and rhs coefficients
        # Uses algorithm proposed here:
        #
            https://stackoverflow.com/questions/42637872/solve-system-of-linear-integer-equations-in-python
        matrix = Matrix(coeff_matrix)
        null_vectors = matrix.nullspace()

        if null_vectors == []:
            raise ValueError("Invalid ReactionT. Reaction cannot be balanced.")

        null_vectors = null_vectors[0]
        multiple = lcm([val.q for val in null_vectors])
        x = multiple * null_vectors
        solution = np.array([int(val) for val in x]).tolist()

        lhs_coeffs = solution[:len(reactants)]
        rhs_coeffs = solution[len(reactants):]

    return lhs_coeffs, rhs_coeffs
```