Assignment 1 Solution

Rithvik Bhogadi and bhogadic

January 29, 2021

This report discusses testing of the `ComplexT` and `TriangleT` classes written for Assignment 1. It also discusses testing of the partner's version of the two classes. The design restrictions for the assignment are critiqued and then various related discussion questions are answered.

1 Assumptions and Exceptions

While creating both modules named ComplexT and TriangleT, some assumptions were made on behalf of the author regarding what input will be used while testing the methods within the module. For example, an assumption that was made while creating the get_r method within the ComplexT module states that irrelevant of the input, the method will always output the result by rounding to two decimal places. This assumption was made to prevent the method from having a longer run time as well as an attempt to reduce the number of decimal places of the result. A very identical assumption was made regarding the get_phi method within the ComplexT module as well. This assumption was made to prevent the method from having a longer run time as well reduce the number of decimal places.

An assumption was made while creating the sqrt method within the ComplexT module. This assumption states that a non-zero value will be used as input while testing. This is because of the fact that the method is unable to find the square root of a negative number. This will prevent the program from running into errors at the run time and allow the program to run smooth. An assumption was made during the creation of the tri_type method within the TriangleT module. This assumption was regarding the fact that the method would only return a single unique type for the valid triangle rather than multiple types due to this method preferring the triangle of right angle type more than the other types. This would mean that irrelevant of the input or how many types of triangle a triangle was to satisfy, if the triangle was a right angle triangle, it would only return that the triangle was to be a right angle triangle.

2 Test Cases and Rationale

The test cases were made for the ComplexT and TriangleT module for which in each test case a specified result would have had to been met in order for the test to have passed. Test cases for each method were made in order to ensure all parts of code of every method were tested. While creating the test cases, each part of the code was made to ensure that the methods were robust and fully working while taking into consideration that a couple of assumptions had to be made. The results of all the test cases have been affirmative and guarantee a full working method while taking into consideration the assumptions that were made. For example, while creating a test case for get_sides method within the TriangleT module, the test case had made sure to firstly check whether any one side of the triangle is less than zero. This is in order to check if the triangle is a valid triangle or not and allow for the get_sides method to work. A test case made for the *perim* method within the TriangleT module consisted of firstly checking whether the perimeter of the triangle would be less than zero. This is important to check because if the triangle was to have a perimeter less than zero, it would mean that the triangle is not valid and that it would have failed the test. Therefore, it is important to check that the triangle does not have a perimeter less than 0.

3 Results of Testing Partner's Code

When running the partner‘s code, there were some complications while testing a few methods. For example, while attempting to test the *conj*, *recip* and *sqrt* method from the ComplexT module of the partner‘s code, all of the tests had shown the same error. This error was regarding the fact that $test_driver.py$ file used did not implement "numpy" under the assumption that it was not needed in order to successfully implement a method. When running the test on the is_valid method within the TriangleT module of the partner‘s code, the method had failed to accomplish the task. This was because the partner had not made sure to check whether all of the sides of the triangle had lengths greater than zero. Due to not ensuring this, the method had failed the test case written for that method.

4 Critique of Given Design Specification

The assignment specifications were straightforward and clear. One of the specifications to admire in regards to this assignment was about the creation of modules such as ComplexT and TriangleT. The reason I say this is due to the assignment giving us an experience in which we able to create multiple methods in order to complete a task. This is great experience for us as we not only better our skills typing in python but learn on how to

better our code by doing repetitive commits and testing. This allows us to look back and reflect on previous drafts of our code. The assignment specification required us to create a $test_driver.py$ file which allowed us to create our own test cases to test our methods. This was highly beneficial as it gave us a taste of what the professional workplace would operate like. By creating our own test cases, it allowed for us to apply our intuition in order to change the method to make it more robust and successful.

Although there were not many flaws regarding the assignment specifications, a drawback would be that there were too many ambiguities involved within this assignment which causes many students including myself to assume a lot. Although the assignment was as clear as possible, it would have been nice to be a bit more clear when in regards to creating some methods. This would only improve the quality of code produced by the students. This can implemented within the design by following how other method specifications have been outlined. This can be done by asking for a certain number of mandatory implementations and then asking for a mandatory output. This would allow for the student to be more focused on accomplishing the task rather than having to assume.

5 Answers to Questions

(a) From the ComplexT module, the selectors are "real", "imag", get_r, get_phi, "conj", "recip", "sqrt". From the TriangleT module, the selectors are get_sides, "perim", "area", is_valid, tri_type.

(b) For the ComplexT module, the two possible state variables are the "real" and "imag" method. For the TriangleT module, the two possible state variables are the *init* method and the tri_type method.

(c) In regards to creating two additional methods within the ComplexT module, it would not make sense to create such methods as it is mathematically impossible to determine whether or not a complex number is greater than or less than the other. Although it is possible to determine whether two complex numbers are equal to each other or not, this does not signal that by changing the specifications of the "equal" method that this would allow for the program to determine which complex number is greater than or less than the other.

(d) In order to check whether a triangle is valid or not within the TriangleT module in regards to the is_valid method we are able to figure out which triangle is valid and which ones aren't valid. This is done by firstly checking to see whether all of the three lengths of the triangle are positive integers greater than zero. This is because it is

impossible for any one of the sides of a triangle to have length of zero or below. If a sample triangle were to pass the first condition then it must check to see whether the addition of any two side of the triangle are greater than the third side. This is a very fundamental rule in creating a valid triangle. If these conditions were to be met then the sample triangle would be considered a valid triangle. Secondly, if a triangle was not valid, then the is_valid method would return false.

(e) It would not be a good idea to have state variable that stores the type of triangle within the tri_type method because there can be the possibility of a triangle fulfilling the requirements to be a triangle of two types. For example, a triangle can be both scalene and right angle triangle. A new variable within the constructor method might not be able to be assigned two types of triangle within one variable. Therefore, it would not be a good idea.

(f) The performance of a product depends upon the qualities such as speed, storage and efficiency. Efficiency depends upon the ability to use computer resources such as memory, time and communication. Hench if the product is not very effiecient it affects the performance as well. The usability of a software product depends on how easy it would be to use that software product. The usability of the software product depends on the capabilities and preferences of the user. Although this were true, if the performance of the product is not good, it adversely affects the usability of the product.

(g) At times although using a rational design process to provide documentation to clients or reviewers maybe be beneficial in order to produce quality software, there are times when it is not neccessary to "fake" the rational design process. For example, if a software developer were developing software for themselves and not for any client or reviewer, then the developer would not have to provide documentation in order to make it seem as of they had followed a rational process. Another example when it is not neccessary to "fake" a rational design process is if there is very small implementation.

(h) The reusability of a component from a software may affect the reliability of a product in two different ways. For example, if the product from which a component had been reused from had flaws within it, then the code that would have been reused to create a new product would consist of those flaws as well. Although this were true, if a component from a certain software product was to be reused in order to create a new product it could at times be used to fix any lingering problems within the new product and improve the software quality as a whole.

(i) An example of how a programming language can be an abstraction built on hardware is found within the assignment specifications. This is because when we are given method specifications, we are required to write lines of python code within an IDE in which we are meant to compile and test the functionality of our methods. When the code is compiled, the IDE will translate the python code into machine code in which hardware will then perform any specific task. This is important because hardware only corresponds to machines code. Therefore, it is possible that a programming language can be an abstraction on top of a hardware.

F Code for complex_adt.py

```
## @file complex_adt.py
#  @author Rithvik Bhogadi
#  @brief Contains a class for working with complex numbers
#  @date 01/21/2021

import math
import cmath

## @brief An ADT for complex numbers
#  @details A complex is composed of a real and imaginary part
class ComplexT:

  ## @brief Constructor for ComplexT
  #  @details Creates a complex number based on a given x and y float values
  #  @param self representing the created ComplexT object
  #  @param x Integer representing the real float value
  #  @param y Integer representing the imaginary float value
  def __init__(self, x, y):
    self.x = x
    self.y = y

  ## @brief Gets the real part of the complex number
  #  @return Integer representing the real part of the complex number
  def real(self):
    return self.x

  ## @brief Gets the imaginary part of the complex number
  #  @return Integer representing the imaginary part of the complex number
  def imag(self):
    return self.y

  ## @brief Gets the absolute value of a complex number
  #  @return Integer representing the modulus of a complex number
  def get_r(self):
    real_r = math.pow(self.x, 2)
    imag_r = math.pow(self.y, 2)
    return math.sqrt(real_r + imag_r)

  ## @brief Returns the phase of a complex number
  #  @return return the phase of a complex number in radians
  def get_phi(self):
    return cmath.phase(complex(self.y, self.x))

  ## @brief compares the equivalence between two complex numbers
  # @param argument The argument of ComplexT
  #  @return returns true or false depending on the comparison between two complex numbers
  def equal(self, argument):
    if argument.x == self.x and argument.y == self.y:
        return True
    else:
        return False

  ## @brief Find the complex conjugate of the complex number
  #  @return returns the complext conjugate of the current object
  def conj(self):
    conj_x = self.x
    conj_y = (self.y * -1)
    return ComplexT(conj_x, conj_y)

  ## @brief Adds two complex numbers together
  #  @param argument A complex number
  #  @return returns the sum of two complex numbers
  def add(self, argument):
    add_x = self.x + argument.x
    add_y = self.y + argument.y
    return ComplexT(add_x, add_y)

  ## @brief subtracts two complex number
  #  @param Argument A complex number
  #  @return return the difference in the form of a complex number
  def sub(self, argument):
    sub_x = self.x - argument.x
    sub_y = self.y - argument.y
    return ComplexT(sub_x, sub_y)

  ## @brief Performs the multiplication between two complex numbers
```

```
#  @param Argument A complex number
#  @return return the product after the multiplication
def mult(self, argument):
  mult_x = self.x
  mult_y = self.y
  multo_x = argument.x
  multo_y = argument.y
  multa = mult_x * multo_x
  multay = mult_y * multo_y
  multa2 = mult_x * multo_y
  multay2 = mult_y * multo_x
  mult_ul = multa - multay
  mult_ul2 = multa2 + multay2
  return ComplexT(mult_ul, mult_ul2)

## @brief Performs the reciprocal of a complex number
#  @return return the reciprocal of a complex number
def recip(self):
  recip_x = self.x
  recip_y = (self.y * -1)
  recip_powx = math.pow(self.x, 2)
  recip_powy = math.pow(self.y, 2)
  recip_x1 = (recip_x/(recip_powx + recip_powy))
  recip_y1 = (recip_y/(recip_powx + recip_powy))
  return ComplexT(recip_x1, recip_y1)

## @brief Performs the division between two complex numbers
#  @param Argument A complex number
#  @return return the product after the multiplication
def div(self, argument):
  div_xy = argument.recip()
  return self.mult(div_xy)

## @brief Performs the square root of a complex number
#  @return return the positive square root of the current object
def sqrt(self):
  sqrt_x = self.x
  sqrt_y = self.y
  sqrt_xy = cmath.sqrt(complex(sqrt_x, sqrt_y))
  return ComplexT(sqrt_xy.real, sqrt_xy.imag)
```

G Code for triangle_adt.py

```
## @file triangle_adt.py
#  @author Rithvik Bhogadi
#  @brief Contains a class for working with Triangles
#  @date 01/21/2021


import math
import cmath
from enum import Enum

## @brief An ADT for different types of triangles
#  @details A TriType is made up a different types of triangles
class TriType(Enum):
    equilat = 1
    isosceles = 2
    scalene = 3
    right = 4

## @brief An ADT for triangles and their components
#  @details A triangle is composed of three lengths
class TriangleT:

  ## @brief Constructor for TriangleT
  #  @details Creates a triangle based on three side lengths
  #  @param l1 Integer representing one of the side lengths of a triangle
  #  @param l2 Integer representing one of the side lengths of a triangle
  #  @param l3 Integer representing one of the side lengths of a triangle
  def __init__(self, l1, l2, l3):
    self.l1 = l1
    self.l2 = l2
    self.l3 = l3

  ## @brief Gets the side lengths of the triangle
  #  @return returns a tuple, where each integer is the length of one side of the triangle
  def get_sides(self):
    return (self.l1, self.l2, self.l3)

  ## @brief checks to see if two triangle are equal to each other
  #  @param argument A second triangle object
  #  @return returns true if both triangles are equal to each other, otherwise it returns false
  def equal(self, argument):
    if self.l1 > 0 and self.l2 > 0 and self.l3 > 0:
      if argument.l1 > 0 and argument.l2 > 0 and argument.l3 > 0:
        if self.l1 == argument.l1 and self.l2 == argument.l2 and self.l3 == argument.l3:
            return True
    else:
        return False

  ## @brief calculates the perimeter of the triangle
  #  @return returns an integer representing the perimeter of the current triangle
  def perim(self):
    length_one = self.l1
    length_two = self.l2
    length_three = self.l3
    sum_one = (length_one + length_two + length_three)
    if length_one > 0 and length_two > 0 and length_three > 0:
        return sum_one
    else:
        return False

  ## @brief calculates the area of the triangle
  #  @return returns a float representing the area of the current triangle
  def area(self):
    length_o1 = self.l1
    length_o2 = self.l2
    length_o3 = self.l3
    semi_p = (length_o1 + length_o2 + length_o3)/2
    ar_o1 = (semi_p * (semi_p - length_o1) * (semi_p - length_o2) * (semi_p - length_o3)) ** 0.5
    if length_o1 > 0 and length_o2 > 0 and length_o3 > 0:
        return ar_o1

  ## @brief checks to see whether the triangle is a valid triangle or not
  #  @return returns true if the triangle is a valid triangle, otherwise it will return false
  def is_valid(self):
    valid_l1 = self.l1
    valid_l2 = self.l2
```

```
    valid_l3 = self.l3
    if valid_l1 > 0 and valid_l2 > 0 and valid_l3 > 0:
        if ((valid_l1 + valid_l2) > valid_l3) and ((valid_l2 + valid_l3) > valid_l1) and ((valid_l1 +
            valid_l3) > valid_l2):
            return True
    else:
        return False

## @brief Checks to see what type of triangle the triangle is
# @details Assuming that this method will only return a single unique type of triangle rather than
    multple as per the syntax
#  @return returns the type of triangle such as equilateral, isosceles, scalene or right angle
    triangle
def tri_type(self):
    tri_l1 = self.l1
    tri_l2 = self.l2
    tri_l3 = self.l3
    do_l1 = math.pow(tri_l1, 2)
    do_l2 = math.pow(tri_l2, 2)
    do_l3 = math.pow(tri_l3, 2)
    if tri_l1 > 0 and tri_l2 > 0 and tri_l3 > 0:
        if tri_l1 == tri_l2 and tri_l2 == tri_l3:
            return TriType.equilat
        lister = [do_l1, do_l2, do_l3]
        lister.sort()
        if ((lister[0] + lister[1]) == lister[2]):
            return TriType.right
        if ((tri_l1 == tri_l3) != tri_l2) or ((tri_l2 == tri_l3) != tri_l1):
            return TriType.isosceles
        if tri_l1 != tri_l2 and tri_l1 != tri_l3 and tri_l2 != tri_l3:
            return TriType.scalene
    else:
        return False
```

H Code for test_driver.py

```
## @file test_driver.py
#  @author Rithvik Bhogadi
#  @brief Tests for complex_adt.py and triangle_adt.py
#  @date 01/21/2021

from complex_adt import ComplexT
from triangle_adt import TriangleT, TriType

## ComplexT
# Test of add method
a = ComplexT(1.0, 2.0)
b = ComplexT(0.5, -0.5)
c = a.add(b)
if(c.x == 1.5 and c.y == 1.5):
    print("add test passes")
else:
    print("add test FAILS")

a = ComplexT(1.0, 2.0)
if(a.x == 1.0):
    print("real test passes")
else:
    print("real test FAILS")

a = ComplexT(3.0, 4.0)
if(a.y == 4.0):
    print("imag test passes")
else:
    print("imag test FAILS")

a = ComplexT(-2.0, 3.0)
getso = round(a.get_r(), 2)
if(getso == 3.61):
    print("get_r test passes")
else:
    print("get_r test FAILS")

a = ComplexT(1.0, 2.0)
phiso = round(a.get_phi(), 2)
if(phiso == 0.46):
    print("get_phi test passes")
else:
    print("get_phi test FAILS")

a = ComplexT(1.5, 1.5)
b = ComplexT(1.5, 1.4)
c = a.equal(b)
if(c == True):
    print("equal test passes")
else:
    print("equal test FAILS")

a = ComplexT(1.0, -2.0)
b = ComplexT(1.0, -2.0)
c = a.conj()
if(c.x == 1.0 and c.y == -2.0):
    print("conj test passes")
else:
    print("conj test FAILS")

a = ComplexT(1.0, 2.0)
b = ComplexT(0.5, 0.5)
c = a.sub(b)
if(c.x == 0.5 and c.y == 1.5):
    print("sub test passes")
else:
    print("sub test FAILS")

a = ComplexT(3.0, -4.0)
b = ComplexT(4.0, 5.0)
c = a.mult(b)
if(c.x == 32.0 and c.y == -1.0):
    print("mult test passes")
else:
    print("mult test FAILS")
```

```
# @details Assuming that non-zero value will be used for input
a = ComplexT(3.0, 4.0)
c = a.sqrt()
if(c.x == 2.0 and c.y == 1.0):
    print("sqrt test passes")
else:
    print("sqrt test fails")

a = ComplexT(2.0, 4.0)
c = a.recip()
if(c.x == 1/10 and c.y == -1/5):
    print("recip test passes")
else:
    print("recip test FAILS")

a = ComplexT(3.0, -1.0)
b = ComplexT(2.0, -2.0)
c = a.div(b)
if(c.equal(ComplexT(1.0, 0.5))):
    print("div test passes")
else:
    print("div test FAILS")


# TriangleT
t1 = TriangleT(3, 4, 5)
t1.get_sides()
if(t1.l1 < 0 and t1.l2 < 0 and t1.l3 < 0):
    print("get_sides test FAILS")
if(t1.l1 == 3 and t1.l2 == 4 and t1.l3 == 5):
    print("get_sides test passes")
else:
    print("get_sides test FAILS")

# Test of equal method
t1 = TriangleT(4, 5, 6)
t2 = TriangleT(4, 5, 7)
if (t1.equal(t2)):
    print("triangle equal test passes")
else:
    print("triangle equal test FAILS")

t1 = TriangleT(4, 6, 2)
if(t1.perim() < 0):
    print("perim test FAILS")
if (t1.perim() == 12):
    print("perim test passes")
else:
    print("perim test FAILS")

t1 = TriangleT(5.0, 7.0, 8.0)
aresa = round(t1.area(), 2)
if(aresa == 17.32):
    print("area test passes")
else:
    print("area test FAILS")

t1 = TriangleT(0, 23, 11)
if(t1.is_valid == True):
    print("is_valid test passes")
else:
    print("is_valid test FAILS")

# Test triangle type
t1 = TriangleT(3, 4, 5)
if (t1.tri_type() == TriType.equilat):
    print("tri_type test passes")
else:
    print("tri_type test FAILS")

if (t1.tri_type() == TriType.isosceles):
    print("tri_type test passes")
else:
    print("tri_type test FAILS")

if (t1.tri_type() == TriType.scalene):
    print("tri_type test passes")
else:
    print("tri_type test FAILS")
```

```
if (t1.tri_type() == TriType.right):
    print("tri_type test passes")
else:
    print("tri_type test FAILS")
```

I Code for Partner's complex_adt.py

```
## @file complex_adt.py
#  @author Shiraz Masliah
#  @title Step 1
#  @date January 28th, 2021

##  @brief Going based off of the assumption that all inputs are correct and computable
#   An assumption that a value of zero will not be supplied

import cmath
import numpy as np

class ComplexT:

        ## @brief ComplexT constructor
        #  @details Initializes ComplexT object with real, imaginary and complex numbers
        #  @params x The real part of the complex number
        #  @params y The imaginary part of the complex number
        def __init__(self, x, y):
                self.x = x
                self.y = y
                self.z = complex(self.x, self.y)

        ## @brief Getter for the real part of the complex number
        #  @return Real part of complex number (x)
        def real(self):
                return (self.x)

        ## @brief Getter for the imaginary part of the complex number
        #  @return Imaginary part of complex number (y)
        def imag(self):
                return (self.y)

        ## @brief Getter for the complex number
        #  @return The complex number
        def get_z(self):
                return (self.z)

        ## @brief Getter for the absolute value of the complex number
        #  @return Absolute value of complex number
        def get_r(self):
                return (abs(self.z))

        ## @brief Returns phase of complex number in radians
        #  @details Ranges from -pi to pi, and rounds to 3 decimal places
        #  @return Phase of complex number
        def get_phi(self):
                phi = cmath.phase(self.z)
                return (round(phi, 3))

        ## @brief Checks equality of two complexTs
        #  @return boolean True or False
        def equal(self, complexT):
                return (self.z == complexT.z)

        ## @brief Returns the conjugate of a complex number
        #  @return the conjugate of the complex number
        def conj(self):
                return (np.conj(self.z))

        ## @brief Adds two complex numbers together
        #  @return the sum of the argument and current object
        def add(self, complexT):
                complex_num = self.z + complexT.z
                return ComplexT(complex_num.real, complex_num.imag)

        ## @brief Subtracts argument from current object
        #  @return the difference of the argument and current object
        def sub(self, complexT):
                complex_num = self.z - complexT.z
                return ComplexT(complex_num.real, complex_num.imag)

        ## @brief Multiplies argument and current object
        #  @return the product of the argument and current object
        def mult(self, complexT):
                complex_num = self.z * complexT.z
                return ComplexT(complex_num.real, complex_num.imag)
```

```
        ## @brief Finds the reciprocal of the current object
        #  and rounds to 3 decimal places
        #  @return the reciprocal of a complex number
        def recip(self):
                return (np.round((np.reciprocal(self.z)),3))

        ## @brief Divides current object by argument
        #  @return the divided complex number
        def div(self, complexT):
                return (self.z / complexT.z)

        ## @brief Finds the square root of the current object
        #  and rounds to 3 decimal places
        #  @return the positive square root of the complex number
        def sqrt(self):
                return (np.round((cmath.sqrt(self.z)),3))
```

J Code for Partner's triangle_adt.py

```
## @file triangle_adt.py
#  @author Shiraz Masliah
#  @title Step 2
#  @date January 21st, 2021

## @details Going based off of the assumption that all inputs are correct and computable

from enum import Enum

class TriType:
        #  @details enum variables
        right = 0
        equilat = 1
        isosceles = 2
        scalene = 3

## @brief This class represents a triangle
#  @details This class represents a triangle as (a, b, c)
#  where each is a side of the triangle
class TriangleT:

        #  @param python convention is to put double underscore before the identifier
        def __init__(self, a, b, c):
        #  private variables, each a side length
                self.a = a
                self.b = b
                self.c = c

        ## @brief Returns the side lengths
        #  @return tuple of integers, each the length of one side of the triangle
        def get_sides(self):
                return (self.a, self.b, self.c)

        ## @brief Checks if two triangles are the same
        #  @return true if TriangleT is equal to the argument
        def equal(self, triangleT):
                selfList = [self.a, self.b, self.c]
                triangleTList = [triangleT.a, triangleT.b, triangleT.c]
                selfList.sort()
                triangleTList.sort()
                return (selfList == triangleTList)

        ## @brief This function calculates the perimeter of the given triangle
        #  @return perimeter of triangle
        def perim(self):
                return (self.a + self.b + self.c)

        ## @brief This function calculates the area of the given triangle
        #  @detail rounded to 3 decimal places
        #  @return area of triangle
        def area(self):
                s = (self.a + self.b + self.c) / 2
                return (round(float((s*(s-self.a)*(s-self.b)*(s-self.c))**0.5),3))
```

```
## @brief This function checks if the triangle is valid by making sure the sum of two sides
## is greater than the third
#  @return True if the 3 sides form a valid triangle
def is_valid(self):
        if self.a + self.b > self.c and self.b + self.c > self.a and self.c + self.a > self.b:
                return True
        else:
                return False

#  @return the type of triangle
#  @param prioritizes right angle triangles
def tri_type(self):
        x, y, z = self.a ** 2, self.b ** 2, self.c ** 2
        if max(x, y, z) == (x + y + z - max(x, y, z)):
                return TriType.right
        elif(self.a == self.b and self.b == self.c and self.a == self.c):
                return TriType.equilat
        elif(self.a == self.b or self.b == self.c or self.a == self.c):
                return TriType.isosceles
        else:
                return TriType.scalene
```