# Assignment 1 Solution

Anika Krishna Peer, 400246282, peera1

January 28, 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

For this assignment there were a few main assumptions made in order to prevent errors from arising during the testing period. These assumptions were made on an individual basis, meaning each method was allowed assumptions based on its need, in order to produce an error-free output. Below are detailed lists of the assumptions made when writing `complex_adt` and `triangle_adt`.

Starting with `complex_adt`:

- In the method `get_phi()`, I made the decision to assume that I would never receive an input for a complex number with both the imaginary and real parts equaling 0. Otherwise, if both the real and imaginary parts of the complex number were 0, the resulting phi would be undefined.
- In the method `recip()`, the same assumption as in `get_phi()` is made. The reason for this is because if both parts (real and imaginary) are 0, there will be an error resulting from dividing by 0.
- In the method `div()` I again made the assumption that both real and imaginary parts would not be zero. I chose to do so because, of the resulting error from dividing by zero.

Now looking at `triangle_adt`:

- The main assumption made was that all inputs were greater than 0. This is part of making a possible triangle if not a valid one.
- I also assumed in `tri_type()` and `area()` that the triangle going to be input would be a valid one. Though in a more complex program it would be more prudent to make sure all the methods had valid triangle inputs, these were specifically important because otherwise they would result in errors.

# 2 Test Cases and Rationale

In order to provide a thorough range of test cases, I decided to test every single method with at least 2 - 3 test cases. The reason I chose to do this was so that I could use these methods in later test cases. Each test case is prefaced with a few lines of creating either `TriangleT` or `ComplexT` objects.

Beginning with `complex_adt`:

- The test cases I selected for most of the getters were random as there were no specific cases that needed to be tested for getters.
- The test cases for `conj()` were a combination of negative real numbers as well as the value 0.0. I chose these values specifically to see if the program could handle making conjugates of complex numbers with negative or zero real values.
- For `add()` I used a mix of positive and negative values for both the imaginary and real parts of the complex numbers. this was essential to ensure that my program handled adding negatives properly. As in `conj()`, I was sure to include zero values.
- It should be noted that this same trend is continued for the rest of the values used to test the other methods in the program, i.e. I used a mix of positive values, negative values, and zeroes.

Most of the tests for `triangle_adt` are fairly simple due to the fact that more work is done with boolean and integer value. Hence there is less room for error margins in these test cases.

- Testing `get_sides` was simply a matter of comparing the tuple results. Without the hinderance of negative and zero values, this was a rather straightforward test and did not require any specific test cases.

- In order to properly test `equal()`, I made use of two triangles that were of the same size but maintained a different order of edges. This allowed me to check if my method was properly checking that all sides of the triangle were equal, without being affected by the order.
- Testing for `perim()` and `area()` forced me to use slightly larger triangles to ensure that I would still get accurate results despite the size. As such, I used triangles with sizes 98, 99, and 100 to bring some variety in size amongst the test cases.
- For `is_valid()`, I decided to test two valid triangles and one invalid one. I chose to use differing sizes in the valid triangles in order to ensure my program could check the validity of triangles no matter their size.
- In terms of `tri_type()`, I simply picked to test all 4 types of triangles to make sure that my program could identify all the individual types.

In terms of the actual comparison of values used to determine if the correct output was received, I used both approximate and equal test cases. For approximate test cases, I rounded outputted values to 3 decimal places and then found the absolute value of the difference between the expected output and the result. This would represent an error calculation. I then picked an arbitrary error 0.00001 which I felt was precise enough for this program, and used it as an upper bound for the calculation. In short, if the error was less than this upper bound, I would consider the expected output and the result approximately equal, otherwise the test case would have failed. I used this on the test cases for methods with outputs showing greater amounts of digits after the decimal. Some of those methods are `mult()`, `recip()` and `sqrt()`. For equal test cases, I used two types of tests. One was used purely for exactly-equal results, such as boolean outputs or tritypes. These results are non-numerical and as such do not need to be approximated. The other test was for nearly-equal results such as addition or subtraction, where the margin of error is much lower and there is no rounding really required. Some of the methods I used this test on are: `conj()` and `add()`.

# 3 Results of Testing Partner's Code

When testing the partner files with my test cases I found that while a lot of the main methods like `add()` and `sub()` had been passed with ease, tests for methods like `sqrt()` and `mult()` failed. A lot of the multi-step methods tended to pass merely half of the test cases. Upon further inspection, I also took note that the test cases that did not pass were significantly different from the expected answers. It was not a matter of decimal places but rather I saw completely different numbers from tests to expected results. Below I have outlined why I believe there were differences for each method.

`complex_adt`:

- `mult()`: An incorrect formula was used. `self.real() * self.y` should be changed to `num_mult.real() * self.y`. This way the real part of `num_mult` will multiply itself by the imaginary value and complete the multiplication as per the formula from the given Wikipedia article on the specification.
- `div()`: An incorrect formula was used. It would be prudent to switch the other of the terms in the imaginary part of this method. Since a subtraction is performed, to find the result of dividing two complex numbers, the order of the terms matter.
- `sqrt()`: An incorrect formula was used. There is a missing part of the formula. There should be a sign function which can attribute positive and negative signs to the output values.

`triangle_adt`:

- `get_sides()`: For this method, the error was simply outputting an array instead of the expected tuple.
- `area()`: An incorrect formula was used. The formula used is supposed to be Heron's formula, which was correctly determined. However, the error is the use of the perimeter over the semi-perimeter.

# 4 Critique of Given Design Specification

One of the advantages of the specification is the lack of redundancy. Every method has a purpose and there is very little overlap between the purposes of each method. This is true for both `triangle_adt` and `complex_adt`. Although in terms of use in the real world, this sort of modularity may be impractical because of the shortness of each individual method, it suited the purpose of testing and documentation well. Another advantage was the clarity of the specification. It was very evident what purpose was served by each method as well as the expected inputs and outputs. This made for better testing and less ambiguity when doing documentation. In order to improve the design, I would change the specification so that certain methods must make use of the other methods in the program in order to be considered complete. For example, when using `mult()` for `complex_adt` one could have reused `add()`. The same could be said about using `mult()` and `recip()` when creating `div()`. Making that explicit in the specification would force one to think what methods have already been created and how that code can be reused, touching upon the concepts we learned in class.

# 5 Answers to Questions

(a) The methods for the classes `ComplexT` which are selectors (getters) are `real()` and `imag()` which simply return the real and imaginary parts of the complex number. The methods `get_r()`, `get_phi()`, `conj()`, `recip()` and `sqrt()` are also getters according to the in-class definition where a getter can calculate a value derived from inspecting a state. As for `TriangleT`, the getter `get_sides()` simply returns all three sides of the triangle. The other getters in this class are `perim()`, `area()`, `is_valid()` and `tri_type()`, which follow the same rules that a getter can calculate a value derived from inspecting a state. There are no mutators (setters) in these classes.

(b) For `ComplexT` the possible instance variables are `self.r` and `self.phi`. The value `self.r` represents the absolute value of the complex number and `self.phi` represents the phase angle. The reason we can use these state variables for this class is that you can get the real and imaginary values from the r and phi values, and vice versa. As for `TriangleT`, the possible instance variables are `self.sides` which represents all 3 sides of the triangle in the form of tuple, similar to the method `get_sides()`, and `self.x`, `self.y`, `self.z` which represent the three sides of the triangle individually as their own variables. As before, the tuple can be used to find the individual sides and vice versa. In `TriangleT` the order of the sides does not matter and has no impact on the calculations or behaviour of the program.

(c) In the current context of the class `ComplexT`, it would make sense to make less than and greater than methods. Having these methods allows the user to make quick work of comparisons between different complex numbers. Additionally, the documentation will make sure that there is a standard method in which the complex numbers are compared, so the person using the comparisons will be sure to receive consistent results if they require reusability, which is one of the software qualities.

(d) It is possible that the three integers input into the constructor for TriangleT will not form a geometrically valid triangle. In this case, the class should allow the invalid input. At the start of each method (aside from getters and setters) `is_valid()` should be run. When it inevitably returns false, a message should be output that the input triangle is invalid and cannot be used in this class. The reason this should occur is because aside from `perim()`and `equal()` the other methods in the class are dependent on the validity of the triangle in order to output an answer that is not erroneous.

(e) If the `TriangleT` class had a state variable for the type of triangle, it could be very beneficial in order to perform calculations for additional methods like finding the angles of each of the vertices. This also removes the need for the enum code because the classification is now done from the state variables. However, it should be noted that using enums is the standard when working with predefined values. Additionally, enums use class syntax, so it is easy to see what the different possiblities are for the types of triangles, and differentiate between all of them. Also, if for some reason, we wanted to assign a numerical value to the types of triangles, that is only possible through enum.

(f) Performance is defined as an external quality that is based on user requirements. Usability, on the other hand, is a quality achieved only by software that users find easy to use. As noted in lecture 4, "Poor performance affects the usability of a product." This is elaborated on in the textbook in terms of multiple examples. One such example denotes that if a software system is very slow, it can reduce the user's productivity and no longer meet their needs. Therefore, the software product is no longer usable because its poor performance has made it difficult for it to be used by human users.

(g) There are no situations where it is not necessary to fake the rational design process, because most projects do not proceed in a planned linear fashion. For example, a project could be at the coding stage and a client could suddenly deliver a new requirement for the software. As this would be an unforseen circumstance, an engineer would probably make changes to the requirements and then implement the newly required functionality. Later, it would be necessary to go back to the problem statement, development plan and other steps and adjust them as needed. An important point to remember is that these changes can sometimes be because of updates in softwares used alongside the one being developed. In short, the importance of the rational design process stems from the fact that software requirements are not fully known when starting a project and can change over time.

(h) Reusability is defined as the ability to take a product and use it to build a new one. Reliability is more about software doing what it is suppossed to do. If a software product is reliable, it can logically be altered and reused. However, a software product that is unreliable is not worth being reused because it does not accomplish its intended purpose, and in the process of trying to reuse it, it could imply the new product will also be unreliable.

(i) When saying programming languages are abstractions built upon hardware, the intended message is that often when using the programming language, one does not need to know what is happening with the inner workings of the hardware in order to have the program work. An example of this could be when a software engineer is trying to improve the speed of one of their computer chips. They do not necessarily need to know how the data and memory will move around the chip as they start to program on it. That is all managed by the programming language itself, which interacts with the hardware in a way that does not provide an open line of visibility to the engineer. This abstraction, however, is very useful because it prevents software engineers from getting lost in the details and confusing themselves with the machine's workings rather than focusing on the programming itself.

Some of the information I used to answer questions and format my report is present at the following links/locations:

- `https://gitlab.cas.mcmaster.ca/smiths/se2aa4_cs2me3/-/blob/master/Lectures/L04_SoftwareQualityContd/SoftwareQualityContd.pdf`
- `https://gitlab.cas.mcmaster.ca/smiths/se2aa4_cs2me3/-/blob/master/Assignments/PreviousYears/2020/A1/A1.pdf`
- Fundamentals of Software Engineering: Second Edition

# F Code for complex_adt.py

```
## @file complex_adt.py
#  @author Anika Krishna Peer
#  @brief Contains a class for working with complex numbers
#  @date 01/16/2020


import math


## @brief An ADT for complex numbers
#  @details A complex number is made of a real and complex part.
class ComplexT:
# Looked at all files here for more than one method:
#
    https://gitlab.cas.mcmaster.ca/smiths/se2aa4_cs2me3/-/tree/master/Assignments/PreviousYears/2019/A1/A1Soln/src
#
    https://gitlab.cas.mcmaster.ca/smiths/se2aa4_cs2me3/-/tree/master/Assignments/PreviousYears/2020/A1/A1Soln/src

        ## @brief Constructor for ComplexT
        #  @details Creates a ComplexT object given a real and complex part of a complex number.
        #  @param x float representing the real part of the complex number.
        #  @param y float representing the imaginary part of the complex number.
        def __init__(self, x, y):
                self.x = x
                self.y = y

        ## @brief gets the real part (x) from the complex number.
        #  @return float representing the real part of the complex number
        def real(self):
                return self.x

        ## @brief gets the imaginary part (y) from the complex number.
        #  @return float representing the imaginary part of the complex number
        def imag(self):
                return self.y

        ## @brief Calculates the absolute value of a complex number and returns
        #  @details Calculates the absolute value of a complex number using the
        #           square root of the sum of squares of the real and complex parts of a complex
            number.
        #  @return returns float absolute value of complex number
        def get_r(self):
        #  The link below was used to figure out pythons square root function
        #  https://www.geeksforgeeks.org/python-math-function-sqrt/
        #  The link below was used to figure out the complex number function
        #  https://www2.clarku.edu/faculty/djoyce/complex/abs.html

                return math.sqrt((self.x**2) + (self.y**2))

        ## @brief calculates phase of a complex number in radians
        #  @details uses arctan and absolute value of a complex number to calculate phase
        #           of a complex number in radians. Assumes that real and imaginary parts are not 0.
        #  @return returns float value of the phase of a complex number in radians
        def get_phi(self):
        #  Writing my assumptions in the details section was an idea I saw by Andrew Rong on the
            discussion board
        #  https://avenue.cllmcmaster.ca/d2l/le/375601/discussions/threads/1493910/View
        #  As for the formula to calculate phi, I used Wikipedia
        #  https://en.wikipedia.org/wiki/Complex_number
                if ((self.y ==0) & (self.x < 0)):
                        return math.pi
                else:
                        return (2 * (math.atan((self.y)/(self.get_r() + self.x))))

        ## @brief Checks if two complex numbers are equal
        #  @details Compares the real and imaginary parts of two complex numbers to see if they are
            equal
        #  @param comp The complex number to check if its the same as our complex number object
        #  @return Boolean True if the complex numbers are equal and false otherwise
        def equal(self, comp):
        #  Used thew3schools link to find the and notation:
            https://www.w3schools.com/python/python_operators.asp
                return (comp.y == self.y) & (comp.x == self.x)

        ## @brief Returns conjugate of a complex number
        #  @return Returns a Complex number that is the conjugate of our current complex number
```

```
def conj(self):
        return ComplexT(self.x, - self.y)

## @brief Calculates the sum of two complex numbers
#  @detail Adds the real parts of the complex numbers together
#          then does the same for the imaginary parts, then returns the resulting complex
   number.
#  @param compNum complex number that will be added to current complex number
#  @return Returns a ComplexT value (complex number) that is the sum of the
#          argument and the current complex number
def add(self, compNum):
#  Used the Khan Academy link below:
#
     https://www.khanacademy.org/math/algebra2/x2ec2f6f830c9fb89:complex/x2ec2f6f830c9fb89:complex-add-sub/v/adding-
        return ComplexT((self.x+compNum.x),(self.y+compNum.y))

## @brief Calculates the difference of two complex numbers and returns that
#  @detail subtracts the real parts of the complex numbers then does the same for
#          the imaginary parts, then returns the resulting complex number.
#  @param compNum complex number that current complex number will be subtracted by
#  @return Returns a ComplexT value (complex number) that is the difference of the
#          argument and the current complex number
def sub(self, compNum):
#  Used the Khan Academy link below:
#
     https://www.khanacademy.org/math/algebra2/x2ec2f6f830c9fb89:complex/x2ec2f6f830c9fb89:complex-add-sub/v/subtrac
        return ComplexT((self.x-compNum.x),(self.y-compNum.y))

## @brief Calculates the product of two complex numbers and returns that
#  @detail Distributes the complex and real parts of he argument and our
#          complex number and multiples them together, then adds all the terms and multiples
#          the two imaginary parts by -1 to compensate for i**2
#  @param compNum complex number that will be multiplied by current complex number
#  @return Returns a ComplexT value (complex number) that is the product of the argument and
     the current complex number
def mult(self, compNum):
#  Used the Khan Academy link below:
#
     https://www.khanacademy.org/math/algebra2/x2ec2f6f830c9fb89:complex/x2ec2f6f830c9fb89:complex-mul/a/multiplying
        term1 = self.x * compNum.x
        term2 = (self.x * compNum.y) + (self.y * compNum.x)
        term3 = -(self.y * compNum.y)
        final_Imag = term2
        final_Real = term1 + term3
        return ComplexT(final_Real, final_Imag)

## @brief Calculates the reciprocal of a complex number and returns it
#  @detail divides the real and imaginary parts by the sums of the squares of the
#          imaginary and real parts and subtracts them from each other. Assuming both real and
#          imaginary are not 0 at the same time.
#  @return Returns a ComplexT value that is the reciprocal of the current complex number.
def recip(self):
#  Used the Wikipedia link below:
#  https://en.wikipedia.org/wiki/Complex_number#Reciprocal_and_division
        term1 = self.x/((self.x ** 2) + (self.y ** 2))
        term2 = self.y/((self.x ** 2) + (self.y ** 2))
        return ComplexT(term1, -term2)


## @brief Calculates the output of the division of two complex numbers and returns it
#  @detail Assuming both real and imaginary are not 0 at the same time.
#          The output is derived through the sum/differences of the products of real and
   imaginary values
#          as well as the sum of squares.
#  @param compNum complex number that current xomplex number will be divided by.
#  @return Returns a ComplexT value that is the output of the division of a complex
#          number and our complex number (self).
def div(self, compNum):
#  Used the Wikipedia link below:
#  https://en.wikipedia.org/wiki/Complex_number#Reciprocal_and_division
        term1 = ((self.x ** 2) + (self.y ** 2))
        term2Mul = (self.x * compNum.x) + (self.y * compNum.y)
        term3Mul = (self.x * compNum.y) - (self.y * compNum.x)
        final_Imag = term3Mul/term1
        final_Real = term2Mul/term1
        return ComplexT(final_Real, final_Imag)

## @brief Calculates the square root of a complex number and return it.
#  @detail Uses the formula with signum, otherwise returns a normal square root.
```

```
#           Also has the provision of retuning a sqaure root of complex numbers with positive
    or negative real parts
#   @return Returns a ComplexT value that is the positive square root of a complex number
def sqrt(self):
#   Used the given Wikipedia link below:
#   https://en.wikipedia.org/wiki/Complex_number#Reciprocal_and_division
#   https://en.wikipedia.org/wiki/Sign_function
        sign = 0
        if (self.y < 0):
                sign = -1
        elif (self.y == 0):
                if (self.x > 0):
                        return ComplexT(math.sqrt(self.x),0)
                else:
                        return ComplexT(0,math.sqrt(-self.x))
        else:
                sign = 1
        gamma = math.sqrt((self.x + (math.sqrt(self.x**2 + self.y**2)))/2)
        sigma_init = sign * math.sqrt((- self.x + (math.sqrt(self.x**2 + self.y**2)))/2)
        return ComplexT(gamma, sigma_init)
```

# G Code for triangle_adt.py

```
## @file triangle_adt.py
#  @author Anika Krishna Peer
#  @brief Contains a class for working with triangles
#  @date 01/18/2020

import math
from enum import Enum, auto

## @brief Enumerated TriType
#  @details A tritype can be isoceles, scalene, equilateral or right
class TriType(Enum):
        equilat = auto()
        isosceles = auto()
        scalene = auto()
        right = auto()

## @brief An ADT representing triangles
#  @details A triangle is composed of  3 sides, x, y, z
class TriangleT:
# All methods inherently assume that inputs are non-negative and greater than 0
# I got the idea for citing sources inside a method from: Farzan Yazdanjou
# Looked here at multiple files for more than one method:
# https://docs.python.org/3/library/enum.html
#
    https://gitlab.cas.mcmaster.ca/smiths/se2aa4_cs2me3/-/tree/master/Assignments/PreviousYears/2019/A1/A1Soln/src
#
    https://gitlab.cas.mcmaster.ca/smiths/se2aa4_cs2me3/-/tree/master/Assignments/PreviousYears/2020/A1/A1Soln/src

        ## @brief constructor for TriangleT
        #  @details Creates a triangle given 3 sides
        #  @param x Integer respresenting the first length
        #  @param y Integer representing the second length
        #  @param z Integer representing the third length
        def __init__(self, x, y, z):
        # Looked at assignment 2 from 2019 for enum
        #
            https://gitlab.cas.mcmaster.ca/smiths/se2aa4_cs2me3/-/blob/master/Assignments/PreviousYears/2019/A2/A2Soln/src/
                self.x = x
                self.y = y
                self.z = z

        ## @brief gets the three sides of the triangle as a tuple
        #  @return tuple of three integers representing the lengths of the triangle
        def get_sides(self):
                return (self.x, self.y, self.z)

        ## @brief checks if a given triangle is equal to the current triangle
        #  @detail Pulls the two tuples of the triangles and sorts them in lists.
        #          Then compares each of the individual elements to see if the two lists are equal
        #  @param newTriangle Triangle that current Triangle will be compares to to see if they are
            equal
        #  @return a boolean value True if they are equal and False if not
        def equal(self, newTriangle):
                useTuple = newTriangle.get_sides()
                selfTuple = self.get_sides()
                list1 = []
                list2 = []
                for i in useTuple:
                        list1.append(i)
                for j in selfTuple:
                        list2.append(j)

                list1.sort()
                list2.sort()

                for i in range(3):
                        if (list1[i] != list2[i]):
                                return False;

                return True;

        ## @brief Calculates the perimeter of a triangle
        #  @detail Adds all lengths to find perimeter of a triangle
        #  @return Integer representing the perimter of the trianle
        def perim(self):
                return (self.x + self.y + self.z)
```

```
## @brief Calculates the area of a triangle
#  @detail Adds all lengths and then divides by two. Assumes triangle is valid.
#          Then uses the rest of heron's formula to find the area.
#  @return float representing the perimter of the triangle
def area (self):
#  Blog post about Heron's formula
#  https://socratic.org/questions/how-do-you-find-the-area-of-a-triangle-with-3-sides-given
        s = float(self.x + self.y + self.z)/2
        area = math.sqrt(s*(s-self.x )*(s-self.y)*(s-self.z))
        return area


## @brief Checks if triangle is valid
#  @detail Adds two lengths and compares them to the third to see if they are greater.
#          If all of the combinations work returns true.
#  @return Boolean value asserting if triangle is valid
def is_valid (self):
        trial1 = ((self.x + self.y) >= self.z)
        trial2 = ((self.x + self.z) >= self.y)
        trial3 = ((self.z + self.y) >= self.x)
        return (trial1 & trial2 & trial3)

## @brief Classifies triangle as one of the tritypes
#  @detail Assumes triangle is valid. Checks if all sides are equal, then if two are.
#          Then it checks if the squares of any of the two sides equal top the square of the
    third.
#          To check for right triangles. It classifies Non-right triangles as Scalene if none
    of
#          the sides are equal
#  @return Tritype that classifies the triangle as scalene, right, equilateral or isosceles
def tri_type (self):
        if ((self.x == self.y) & (self.y == self.z)):
                return TriType.equilat


        selfTuple = self.get_sides()
        list1 = []
        for i in selfTuple:
                list1.append(i)
        list1.sort()
        for i in range(3):
                if ((i-1) != -1):
                        if (list1 [i] == list1 [i-1]):
                                return TriType.isosceles

        trial1 = ((self.x**2 + self.y**2) == self.z**2)
        trial2 = ((self.x**2 + self.z**2) == self.y**2)
        trial3 = ((self.z**2 + self.y**2) == self.x**2)

        if (trial1 | trial2 | trial3):
                return TriType.right


        if ((self.x != self.y) & (self.y != self.z) & (self.x != self.z)):
                return TriType.scalene
```

# H Code for test_driver.py

```
## @file test_driver.py
#  @author Anika Krishna Peer
#  @brief A module that tests the first module (complex_adt.py) and the second one (triangle_adt.py)
#  @date January 19 2021

# TESTCASES:
# I took the testing method from an old test case for A1 in 2018 as well as an online python float
    comparison document:
#
    https://gitlab.cas.mcmaster.ca/smiths/se2aa4_cs2me3/-/blob/master/Assignments/PreviousYears/2018/A1/A1Soln/src/testSeqs
# https://www.linuxtopia.org/online_books/programming_books/python_programming/python_ch07s02.html

from complex_adt import ComplexT
from triangle_adt import TriangleT, TriType
############################__COMPLEX NUMBER CASES__######################
def equalityCases(test, result, name):
        if (abs(test-result)<0.00001):
                print("Test passed: "+ str(test) + " == " + str(result) +", "+ name)
        else:
                print("Test failed: "+ str(test)+ " != " + str(result) +", "+name)

def equalityCasesBoolean(test, result, name):
        if (test == result):
                print("Test passed: "+ str(test) + " == " + str(result) +", "+ name)
        else:
                print("Test failed: "+ str(test)+ " != " + str(result) +", "+name)

def approxCases(test, result, name):
        if (abs(round(test,3)-result)<=0.00001): # picked the rounding as a test method because
            rounding within 3 decimal places should be more or less accurate.
                print("Test passed: "+ str(test) + " is approximately " + str(result) +", "+ name)
        else:
                print("Test failed: "+ str(test)+ " is not approximately " + str(result) +", "+name)
#real() test cases
def real_test():
        a = ComplexT(1.0, 2.0)
        b = ComplexT(-0.5, 0.5)


        equalityCases(a.real(), 1.0, "No Negatives real part")
        equalityCases(b.real(), -0.5, "Negative real part")

def imag_test():
        a = ComplexT(1.0, 2.0)
        b = ComplexT(0.5, -0.5)

        equalityCases(a.imag(), 2.0, "No negatives real part")
        equalityCases(b.imag(), -0.5, "Negative imaginary part")

def get_r_test():
        a = ComplexT(0.0, 2.4422)
        b = ComplexT(-3.7899, -0.232323)
        approxCases(a.get_r(), 2.442, "Absolute value with 0 part")
        approxCases(b.get_r(), 3.797, "getting absolute value")

def get_phi_test():
        # As stated in the file complex_adt.py, this does not account for entries where both x and y
            are 0
        # As such I will not be testing for such cases.
        a = ComplexT(-1.0003, 0.0)
        b = ComplexT(7.4343, 0.0)
        c = ComplexT(-9.24, 3.33)
        approxCases(a.get_phi(), 3.142, "Phi value with negative real and imag 0 pi")
        approxCases(b.get_phi(), 0.000 , "Phi value positive real number and y = 0")
        approxCases(c.get_phi(), 2.796, "Phi value negative real number and y!= 0")
def equal_test():
        a = ComplexT(-1.0003, 2.0)
        b = ComplexT(-1.0003, 2.0)
        c = ComplexT(1.0003, -2.0)
        d = ComplexT(1.003, 2.03)
        equalityCasesBoolean(a.equal(c), False, "Switching negatives and checking equality")
        equalityCasesBoolean(a.equal(d), False, "Checking equality through decimals")
        equalityCasesBoolean(a.equal(b), True, "Checking equality normally")
def conj_test():
        a = ComplexT(-1.0003, 5.7).conj()
        b = ComplexT(0.0, 3.99).conj()
```

```
        equalityCases(a.real(),-1.0003,"Checking conjugate real part")
        equalityCases(a.imag(),-5.7, "Checking imaginary part of conjugate")
        equalityCases(b.real(),0.0,"Checking conjugate real part")
        equalityCases(b.imag(),-3.99, "Checking imaginary part of conjugate")

def add_test():
        a = ComplexT(-1.0, 2.0)
        b = ComplexT(0.0, -0.5)
        d = ComplexT(0.220, 0.5346)
        e = ComplexT(0.558, 0.3)

        equalityCases(a.add(b).real(),-1.0,"Adding with random negatives - real")
        equalityCases(a.add(b).imag(),1.5,"Adding with random negatives - imag")
        equalityCases(d.add(e).real(),0.778,"Adding with all positives - real")
        equalityCases(d.add(e).imag(),0.8346,"Adding with all positives - imag")

def sub_test():
        a = ComplexT(-1.0, 2.0)
        b = ComplexT(0.0, -0.5)
        d = ComplexT(0.220, 0.5346)
        e = ComplexT(0.558, 0.3)
        equalityCases(a.sub(b).real(),-1.0,"Subtracting with random negatives - real")
        equalityCases(a.sub(b).imag(),2.5,"Subtracting with random negatives - imag")
        equalityCases(d.sub(e).real(),-0.338,"Subtracting with all positives - real")
        equalityCases(d.sub(e).imag(),0.2346,"Subtracting with all positives - imag")

def mult_test():
        a = ComplexT(-1.0, 6.0)
        b = ComplexT(0.0, -5.5)
        d = ComplexT(0.524, 0.83556)
        e = ComplexT(0.55865, 0.4566)
        approxCases(a.mult(b).real(),33,"Multiplying with random negatives - real")
        approxCases(a.mult(b).imag(),5.5,"Multiplying with random negatives - imag")
        approxCases(d.mult(e).real(),-0.089,"Multiplying with all positives - real")
        approxCases(d.mult(e).imag(),0.706,"Multiplying with all positives - imag")

def recip_test():
        # Maintain the assumption that both values cannot be 0 simultaneously.
        a = ComplexT(0.0, -6.0)
        e = ComplexT(0.55865, 0.4566)
        approxCases(a.recip().real(),0,"Reciprocal with random negatives and a 0- real")
        approxCases(a.recip().imag(),0.167,"Reciprocal with random negatives and a 0 - imag")
        approxCases(e.recip().real(),1.073,"Reciprocal with all positives - real")
        approxCases(e.recip().imag(),-0.877,"Reciprocal with all positives - imag")

def div_test():
        # Maintain the assumption that both values cannot be 0 simultaneously.
        a = ComplexT(-3.577, -6.966)
        c = ComplexT(55.99, -3.415555)
        e = ComplexT(0.55865, 0.4566)
        approxCases(c.div(a).real(),-0.056,"Division with both negatives - real")
        approxCases(c.div(a).imag(),-0.128,"Division with both negatives - imag")
        approxCases(e.div(c).real(),57.089,"Division with both negatives - real")
        approxCases(e.div(c).imag(),-52.775,"Division with both negatives - imag")
        approxCases(a.div(e).real(),-0.084,"Division with one set of negatives- real")
        approxCases(a.div(e).imag(),0.037,"Division with one set of negatives- imag")

def sqrt_test():
# In this case the imaginary value cannot
        a = ComplexT(3.577, 0)
        b = ComplexT(-3.577, 0)
        c = ComplexT(55.99, -3.415555)
        e = ComplexT(0.55865, 0.4566)
        approxCases(a.sqrt().real(),1.891,"Square Root test case real.")
        approxCases(a.sqrt().imag(), 0 , "Square Root test case imag.")
        approxCases(b.sqrt().real(),0,"Square Root test case real.")
        approxCases(b.sqrt().imag(),1.891, "Square Root test case imag.")
        approxCases(c.sqrt().real(),7.486,"Square Root test case real.")
        approxCases(c.sqrt().imag(),-0.228,"Square Root test case imag.")
        approxCases(e.sqrt().real(),0.800,"Square Root test case real.")
        approxCases(e.sqrt().imag(),0.285,"Square Root test case imag.")


##########################__TRIANGLE_CASES__####################


def get_sides_test():
        t1 = TriangleT(3, 4, 5)
```

```
        t2 = TriangleT(1,2,3)
        equalityCasesBoolean(t1.get_sides(),(3,4,5),"checking all 3 sides")
        equalityCasesBoolean(t2.get_sides(),(1,2,3),"checking all 3 sides")


def equal_tri_test():
        t1 = TriangleT(3, 4, 5)
        t2 = TriangleT(5, 3, 4)
        t3 = TriangleT(99, 3, 4)
        equalityCasesBoolean(t1.equal(t2),True,"All sides are equal")
        equalityCasesBoolean(t2.equal(t3),False,"All sides are not equal")

def perim_test():
        t1 = TriangleT(7, 9, 5)
        t3 = TriangleT(99,100, 98)
        t2 = TriangleT(4, 7, 8)
        equalityCasesBoolean(t1.perim(),21,"Testing perimeter 1")
        equalityCasesBoolean(t3.perim(),297,"Testing perimeter 2")
        equalityCasesBoolean(t2.perim(),19,"Testing perimeter 3")

def area_test():
        t1 = TriangleT(7, 9, 5)
        t3 = TriangleT(99,100, 98)
        t2 = TriangleT(4, 7, 8)
        approxCases(t1.area(),17.412,"Testing area 1")
        approxCases(t3.area(),4243.091,"Testing area 2")
        approxCases(t2.area(),13.998,"Testing area 3")

def is_valid_test():
        t1 = TriangleT(7, 9, 50000000)
        t2 = TriangleT(4, 7, 8)
        t3 = TriangleT(9,100, 98)
        equalityCasesBoolean(t1.is_valid(), False,"Validity test 1")
        equalityCasesBoolean(t3.is_valid(), True,"Validity test 2")
        equalityCasesBoolean(t2.is_valid(), True,"Validity test 3")
def tri_type_test():
        t1 = TriangleT(7, 7, 7)
        t2 = TriangleT(3, 4, 5)
        t3 = TriangleT(9, 9, 8)
        t4 = TriangleT(7, 9, 13)
        equalityCasesBoolean(t1.tri_type(),TriType.equilat,"Triangle type test equilateral")
        equalityCasesBoolean(t2.tri_type(), TriType.right,"Triangle type test right")
        equalityCasesBoolean(t3.tri_type(), TriType.isosceles,"Triangle type test isosceles")
        equalityCasesBoolean(t4.tri_type(), TriType.scalene,"Triangle type test scalene")




def allTests():
        print("COMPLEX TESTS")
        print("")
        print("")
        real_test()
        imag_test()
        get_r_test()
        get_phi_test()
        equal_test()
        conj_test()
        add_test()
        sub_test()
        mult_test()
        recip_test()
        div_test()
        sqrt_test()
        print("")
        print("")
        print("TRIANGLE TESTS")
        print("")
        print("")
        get_sides_test()
        equal_tri_test()
        perim_test()
        area_test()
        is_valid_test()
        tri_type_test()
allTests()
```

# I Code for Partner's complex_adt.py

```
## @file complex_adt.py
#  @author Samia Anwar
#  @brief Contains a class to manipulate complex numbers
#  @Date January 21st 2021

import math
import numpy

## @brief An ADT for representing complex numbers
#  @details The complex numbers are represented in the form x + y*i
class ComplexT:
        ## @brief Constructor for ComplexT
        #  @details Creates a complext number representation based on given x and
        #                         y assuming they are always passed as real numbers. Real numbers
        #                         are in the set of complex numbers, therefore, y can be 0.
        #  @param x is a real number constant
        #  @param y is a real number coefficient of the square root of  -1.

        def __init__(self, x, y):
                self.x = x
                self.y = y

        ## @brief Gets the constant x from a ComplexT
        #  @return A real number representing the constant of the instance
        def real(self):
                return self.x

        ## @brief Gets the constant x from a ComplexT
        #  @return A real number representing the coefficient of the instance
        def imag(self):
                return self.y

        ## @brief Calculates the absolute value of the complex number
        #  @return The absolute value of the complex number as a float
        def get_r(self):
                self.abs_value = math.sqrt(self.x*self.x + self.y*self.y)
                return self.abs_value

        ## @brief Calculates the phase value of the complex number
        #  @details Checks for the location of imaginary number on the real-imaginary
        #                 plane, and performs the corresponding quadrant calculation
        #  @return The phase of the complex number as a float in radians
        def get_phi(self):
                if self.x > 0:
                        self.phase = numpy.arctan(self.y/self.x)
                elif self.x < 0 and self.y >= 0 :
                        self.phase = numpy.arctan(self.y/self.x) + math.pi
                elif  self.x < 0 and self.y < 0:
                        self.phase = numpy.arctan(self.y/self.x) - math.pi
                elif self.x == 0 and self.y > 0:
                        self.phase = math.pi/2
                elif self.x == 0 and self.y < 0:
                        self.phase = -math.pi/2
                else:
                        self.phase = 0
                return self.phase

        ## @brief Checks if a different ComplexT object is equal to the current one
        #  @details Compares the real and imaginary compoenets of the two instances
        #  @param Accepts a ComplexT object, arg
        #  @return A boolean corresponding to whether or not the two specified
        #          objects are equal to one another, True for they are equal and False otherwise
        def equal(self, arg):
                self._argx = arg.real()
                self._argy = arg.imag()
                return self._argx == self.x and self._argy == self.y

        ## @brief Calculates the conjunct of the imaginary number
        #  @return A ComplexT Object corresponding to the conjunct of the specific instance
        def conj(self):
                return ComplexT (self.x, - self.y)

        ## @brief Adds a different ComplexT object to the current object
        #  @details Adds the real and imaginary components of the two instances
        #  @param Accepts a ComplexT object, num_add
        #  @return A ComplexT object corresponding to the sum of the real and imaginary
```

```
#           and imaginary components
def add(self, num_add):
        self._newx = num_add.real() + self.x
        self._newy = num_add.imag() + self.y
        return ComplexT (self._newx, self._newy)

## @brief Subtracts a different ComplexT object from the current object
#  @details Individually subtracts the real and imaginary components of the two instances
#  @param Accepts a ComplexT object, num_sub
#  @return A ComplexT object corresponding to the difference of the real and imaginary
#           and imaginary components
def sub(self, num_sub):
        self._lessx = self.x - num_sub.real()
        self._lessy = self.y - num_sub.imag()
        return ComplexT (self._lessx, self._lessy)

## @brief Multiplies a different ComplexT object with the current object
#  @details Arithmetically solved formula for (a + b*i) * (x + y*i) and seperated
#                          the constant (a*x - y*b) and the coefficient (b*x + a*y)
#  @param Accepts a ComplexT object, num_mult which acts as a multiplier (a + bi)
#  @return A ComplexT object corresponding to the product of two multipliers
def mult(self, num_mult):
        self._multx = num_mult.real() * self.x - self.y * num_mult.imag()
        self._multy = num_mult.imag() * self.x + self.real() * self.y
        return ComplexT (self._multx, self._multy)

## @brief Calculates the reciprocal or inverse of the complex number
#  @details The formula was retrieved from www.suitcaseofdreams.net/Reciprocals.html
#  @return A ComplexT object corresponding to the reciprocal of the current number
def recip(self):
        if self.x == 0 and self.y == 0:
                return "The reciprocal of zero is undefined"
        else:
                self._recipx = self.x / (self.x * self.x + self.y * self.y)
                self._recipy = - self.y / (self.x * self.x + self.y * self.y)
                return ComplexT(self._recipx, self._recipy)

## @brief Divides a given complex number from the current number
#  @details The formula was retrieved from
    www.math-only-math.com/divisio-of-complex-numbers.html
#  @param An object of ComplexT which acts as the divisor to the current dividend
#  @return A ComplexT Object corresponding to the quotient of the current number over the input
def div(self, divisor):
        self._divx = divisor.real()
        self._divy = divisor.imag()
        if self._divx == 0 and self._divy == 0:
                return "Cannot divide by zero"
        else:
                return ComplexT ( (self.x*self._divx + self.y*self._divy)
                                             / (self._divx * self._divx +
                                                 self._divy*self._divy),
                                             (self.y * self._divx - self._divy * self.x)
                                             / (self._divx * self._divx +
                                                 self._divy*self._divy))
## @brief Calculates the square root of the current ComplexT object
#  @details The formula was retrieved from Stanley Rabinowitz's paper "How to find
#           the Square Root of a Complex Number" published online, found via google search
#  @return A ComplexT object corresponding to the square root of the current number
def sqrt(self):
        self._sqrtx = math.sqrt((self.x) + math.sqrt(self.x*self.x + self.y*self.y)) /
            math.sqrt(2)
        self._sqrty = math.sqrt(math.sqrt(self.x*self.x + self.y*self.y) - self.x) /
            math.sqrt(2)
        return ComplexT (self._sqrtx, self._sqrty)
```

# J Code for Partner's triangle_adt.py

```
## @file triangle_adt.py
#  @author Samia Anwar anwars10
#  @brief
#  @date January 21st, 2021
from enum import Enum
import math
## @brief An ADT for representing individual triangles
```

```python
#  @details The triangle are represented by the lengths of their sides
class TriangleT:
        ## @brief Constructor for Triangle T
        #  @details Creates a representation of triangle based on the length of its sides,
        #           I have assumed the inputs to be the set of real numbers not including zero.
        #  @param The constructor takes 3 parameters corresponding to the three sides of a triangle
        def __init__(self, s1, s2, s3):
                self.s1 = s1
                self.s2 = s2
                self.s3 = s3
        ## @brief Tells the user the side dimensions of the triangle
        #  @return An array of consisting of the length of each side
        def get_sides(self):
                return [self.s1, self.s2, self.s3]

        ## @brief Tells the user if two TriangleT objects are equal to one another
        #  @param Accepts a TriangleT type to compare with the current values
        #  @return A boolean type true for the two are the same and false otherwise
        def equal(self, compTri):
                return set(self.get_sides()) == set(compTri.get_sides())

        ## @brief Tells the user the sum of all the sides of the triangle
        #  @return An num type representing the perimetre of the triangle
        def perim(self):
                return (self.s1 + self.s2 + self.s3)

        ## @brief Tells the user the area of the TriangleT referenced
        #  @return A float representing the are of the TriangleT referenced
        def area(self):
                if self.is_valid() :
                        return math.sqrt(self.perim() * (self.perim() - self.s1) * (self.perim() -
                            self.s2) * (self.perim() - self.s3) )
                else:
                        return 0

        ## @brief Tells the user if the triangle referenced is valid
        #  @details Determines the validity of the triangle based on the sides
        #  @return A boolean value which is true if the triangle is valid, false otherwise
        def is_valid(self):
                if (((self.s1 + self.s2) > self.s3) and ((self.s1 + self.s3) > self.s2) and ((self.s2
                    + self.s3) > self.s1)):
                        return True
                else:
                        return False
        ## @brief Tells the user one name for the type of triangle TriangleT referenced
        #  @details This program prioritises right angle triangle over the others, so
        #           if the triangle is right, it will give only a right angle result and
        #                   not isoceles or scalene.
        #  @return An instance of the TriType class corresponding to right/equilat/isoceles/or scalene
        def tri_type(self):
                if (round(math.sqrt(self.s1 * self.s1  + self.s2 * self.s2)) == round(self.s3)
                        or round(math.sqrt(self.s1 * self.s1 + self.s3 * self.s3)) == round(self.s2)
                        or round(math.sqrt(self.s3 * self.s3 + self.s2 * self.s2)) == round(self.s1)):
                        return TriType.right
                elif (self.s1 == self.s2 and self.s2 == self.s3):
                        return TriType.equilat
                elif(self.s1 == self.s2 or self.s1 == self.s3 or self.s2 == self.s3):
                        return TriType.isoceles
                else:
                        return TriType.scalene

## @brief Creates an enumeration class to store the type of triangle to be referenced by
#         tri_type method within TriangleT
class TriType(Enum):
        equilat = 1
        isoceles = 2
        scalene = 3
        right = 4
```