Assignment 1 Solution

Cassidy Baldin, baldic1

January 27, 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

No exceptions were thrown in this assignment, but some assumptions were made in the process of making these classes written for this assignment. These assumptions are made in the code where applicable, which can also be seen below as:

- Input values passed to the `ComplexT` constructor, were assumed to be of type `float`, and that both input parameters would not be equal to 0 simultaneously (input will never be z = 0 + 0i),
- Return values of `get_phi` were assumed to be in the range of (-pi, pi],
- Input values for method `equal` in class `ComplexT` is of type `ComplexT`, and values are considered to be equal if both the real and imaginary values of the argument are equal to the current real and imaginary values respectively (within 9 decimal places),
- Return value of `conj`, `add`, `sub`, `mult`, `recip`, `div` and `sqrt` is a new ComplexT that is equal to the result of their respective methods,
- Return value of `sqrt` is only the positive part of the square root of the current object,
- Input values passed to the `TriangleT` constructor, were assumed to be positive, non-zero integer values (to relate to the real world application of triangle dimensions),

- Input values for method `equal` in class `TriangleT` is of type `TriangleT`, and values are considered to be equal all side lengths are equal (within 9 decimal places),
- Input value for method `area` is a valid triangle so that the area can actually be calculated,
- Priority labels for method `tri_type` (when a given triangle can have more than one label) is in the order right, equilateral, isosceles, then scalene
- Input value for method `tri_type` is a valid triangle (so that it can be classified accordingly, as there is no option for it to be of type `notvalid`).

2 Test Cases and Rationale

Tests were written such that each method that was implemented into the design had an appropriate amount of test cases that I felt covered the edge/boundary cases for each method respectively. The breakdown of all test cases and rationale are below:

For class `ComplexT`:

To test method `real` and `imag`, I just tested one case for each, as I figured it could either get the required self.variable, or it could not. Since these were basic methods, I did not think that there would be much variance in the results of it.
To test method `get_r`, I used one test case with a right angled triangle, as the absolute value of the complex number would contain a perfect square number. This was a basic case. The other case I tested was to check if the `get_r` method could handle inputs which were negative and find its absolute value

To test method `get_phi`,

To test method `equal`,

To test method `conj`,

To test method `add`,

To test method `sub`,

To test method `mult`,

To test method `recip`,

To test method `div`,

To test method `sqrt`,

For class `TriangleT`:

To test method `get_sides`,

To test method `equal`,

To test method `perim`, I just tested one case, as it was just basic addition of three positive integer numbers, which I did not think would give much variance in the results of different test cases.

To test method `area`,

To test method `is_valid`,

To test method `tri_type`,

3 Results of Testing Partner's Code

4 Critique of Given Design Specification

There were many strengths of the design specification. Firstly, the methods in the class `ComplexT` well defined the aspects of a complex number, and included a variety of methods that allow the user to use the class for many different applications when dealing with complex numbers. Also, the input types for the class being of type `float` is a strength it has, as it allowed a wider range of elements that can be input into the class. In terms of the methods that were implemented, most returned float values as members of a new `ComplexT` object created from certain formulaic calculations, so if the type of this class was not float, and were for example integers instead, then there would need to be rounding to meet the specification, which would cause some inaccuracies in the object. For the `TriangleT` class, it similarly had methods that were meant for dealing with triangles in many ways. The use of the enumerated class `TriType` was also a strength to the design, that allowed the classification of the object triangle for the user from a set of different triangle types to aid the user in creating their triangle object.

The areas of the design specification that could use improvement were in the class `ComplexT`, as in its current implementation based on assumptions it does not account for the complex number being of the form z = 0 + 0i, which technically should be a part of the complex numbers in a mathematical sense. This would currently be a problem for the `recip` and `div` methods, as it would cause a division by zero error. This could be dealt with by adding exceptions to these methods, where it could output an error message to the user, telling them that this error would occur, and therefore that method for that particular input would be invalid. Another area that could use some improvement could be with the `is_valid` method in the `TriangleT` class. In the current implementation, this is used to see if the values input into the constructor for a valid triangle. Two methods in this class assume that the input forms a valid triangle in its current implementation and therefore assume that the input to the constructor is also valid, removing the need for the `is_valid` method. This could have been specified by throwing an exception in these classes using this method instead, or used in the constructor class to ensure validity of the input, which would make the class function better and have less redundancy. Other improvements that could have been made to these classes is the addition of a `get_quadrant` and a `get_height` method for each respective class, as it would allow for more functionality of the objects, as these are important qualities to know about each class. Finally, the design specifications listed could have been less ambiguous, as it allowed for many assumptions or exceptions to be made in the design process which could have been specified more to get a clearer result.

5 Answers to Questions

(a) Methods that are mutators are defined as methods that change the state of the current object, while selectors simply access the value that was set either by the constructor, or a mutator method.

For both classes, there is no instance of the value of self.variable being changed to a new value in any method, therefore in this implementation there are no mutators (setters).

For the class `ComplexT`, the methods that are selectors (getters) are `real`, `imag`, `get_r`, `get_phi`, `conj`, `recip`, and `sqrt`. This is because all of these methods return a value based on the current state of the object they are dealing with. For example, for the `real` method, it returns the real value of the complex object directly, simply reading the value of self.x and returning that. Similarly, for the `get_r` method, it returns the absolute value of the complex number, based on the variables self.x and self.y. Since these directly return a variable or value from the state of the class, it is considered a selector.

For the class `TriangleT`, the methods that are selectors (getters) are `get_sides`, `perim` and `area`. This is because it returns the value of the three sides that were input to the constructor (self.x, self.y and self.z) and does not changes the value of these terms. You could also consider `tri_type` to be a selector, as it returns the type of triangle that the constructor specified, using the variables in the enumerated class `TriType`. Since it returns a variable of that class, it may also be considered a selector.

(b) Two options for state variables for `ComplexT` could be `quadrant` of the complex number, and `magnitude`. If using both of these as state variables, the user of the class could input the quadrant that the complex number is in, as well as its magnitude and that would create the object along with the current real and imaginary state variables. From an implementation standpoint it would be strange, but it would be possible.

Two options for state variables for `TriangleT` could be `height` and `hypotenuse`. This is because...

(c) The class `ComplexT` has an equal method, but I do not think that it would make much sense to have methods for greater than or less than, as in this context, I am not sure how you would end up defining what range of value would be included in those definitions. For example, in the `equal` method, the two complex numbers defined by the function are equal if and only if the real and imaginary parts are both equal respectively. If you wanted to define the method for greater than, you could consider the case that only the real or imaginary value is greater than the current object, but since the complex number represents a value that has two parts to it and acts more like a vector than an integer or a float (which only has 2D components), I do not think that this definition would make much sense. Since the complex numbers do act like vectors, you could define a greater than or less than method to work off of the value of the absolute value of the number (or the magnitude), as this is a float value and can easily be compared between two complex numbers, but this may not reflect if the complex number is greater/less than, only that its absolute value is greater/less than, which is not the same thing. Overall, if you wanted to make the greater/less than methods using the same format as the current implementation where the real and imaginary values of the complex number are compared, then adding these methods in my opinion do not make much sense.

(d) Since in this assignment we were asked to make a method called `is_valid` to check if a triangle was valid, it is very possible that the three integers input to the constructor for `TriangleT`will not form a geometrically valid triangle (and that was exactly what the is_valid method was meant to check). In the case that the input is invalid, I think that the class should not allow the triangle to be constructed (it should throw an error as the first check in the constructor method), as it is not able to actually be a fully formed triangle in the traditional sense. If the triangle that was input is not valid, most of the methods included in the class do not make much sense, as if it cannot physically form a triangle, it will have no perimeter or area as these methods can only apply to a closed shape according to their mathematical definitions and cannot be classified as a triangle in the method tri_type as it is not any of those types of triangles. In my implementation, I assumed that the input would be a valid triangle for the area, so that the result could not be zero or a negative value, which would not make sense for a physical triangle. I also assumed this for the tri_type method, as I did not want the result to not return the value None, as some inputs would not classify to one of the four types in the class TriType, and would therefore return that the input was of type None, which also does not make sense. For these reasons, this is what I think should happen in the case that the given input is an invalid triangle.

(e) If you introduced a state variable for the type of triangle, that would require the user of the class to know what type of triangle it was when inputting values into the class. This might be a good thing, as you could then have an easy way of knowing the height of the triangle if the type was right-angled, which the user of the class might want to know, and it would negate the use of the TriType enumerated class, and the tri_type method, as this would be user input. However, it might be a bad thing to add, as the user might input some values for the side lengths that may not correspond to the type of triangle it actually is, which might cause some issues in the class and with the data they are using the class with, so there may need to be a check to make sure the type is correct based on the given side lengths, but then you would need those methods that I said could be removed previously under this implementation, making the addition of this state variable redundant and unnecessary. These are some reasons why the addition of a state variable for the type of triangle might be a good or a bad idea, depending on the uses, needs and implementation of the class.

(f) (f) relationship between software performance and usability?

(g) (g) are there situations where it is not really necessary to "fake" a rational design process?

(h) (h) how might reusability affect the reliability of products?

(i) (i) what are some examples of how programming languages are abstractions built on top of hardware?

F Code for complex_adt.py

```
## @file complex_adt.py
#  @author Cassidy Baldin
#  @brief Contains a class for creating a complex number
#  @date January 21st, 2021

import math

## @brief An ADT for complex numbers
#  @details A complex number of form  z = x + yi, given a real value x, and an
#           imaginary value y. Uses equations from this source:
#           https://en.wikipedia.org/wiki/Complex_number
class ComplexT:

    ## @brief Constructor for ComplexT
    #  @details Creates a complex number of form  z = x + yi
    #           given real value x, and an imaginary value y. It assumes
    #           that input values are float values, and that the input will
    #           never be z = 0 + 0i.
    #  @param x Float representing real value of complex number
    #  @param y Float representing imaginary value of complex number
    def __init__(self, x, y):
        self.x = x
        self.y = y

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

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

    ## @brief Gets absolute value (or modulus/magnitude) of the complex number
    #  @return Float representing absolute value of the complex number
    def get_r(self):
        return math.sqrt(self.x**2 + self.y**2)

    ## @brief Gets the argument (or phase) of the complex number in radians
    #  @details It is also assumed here that the range of values is
    #           between (-pi, pi]
    #  @return Float representing the phase of the complex number in radians
    def get_phi(self):
        if (self.x < 0 and self.y == 0):
            return math.pi
        else:
            arg = self.y/((math.sqrt(self.x**2 + self.y**2)) + self.x)
            return 2*(math.atan(arg))

    ## @brief Checks if argument and current object are equal
    #  @details They are considered equal if both the real and imaginary
    #           values are equal (within 9 decimal places) to the current real
    #           and imaginary values respectively. It assumes the input is of
    #           the type ComplexT.
    #  @param e ComplexT to compare to current object
    #  @return True if the argument and the current object are equal
    def equal(self, e):
        if math.isclose(e.x, self.x, abs_tol = 0.00000001):
            if (math.isclose(e.y, self.y, abs_tol = 0.00000001)):
                return True
            return False
        return False

    ## @brief Gets the complex conjugate of the current object
    #  @details It assumes that it makes a new ComplexT that is the conjugate
    #           of the current object
    #  @return The complex conjugate of the current object as a ComplexT
    def conj(self):
        new_y = self.y*(-1)
        return ComplexT(self.x, new_y)

    ## @brief Adds argument object to current object
    #  @details It assumes the input is of type ComplexT and makes a new
    #           ComplexT that is the addition of the argument and current object
    #  @param a ComplexT to add to current object
```

```
    #  @return The addition of the current object and argument as a ComplexT
    def add(self, a):
        new_x = self.x + a.x
        new_y = self.y + a.y
        return ComplexT(new_x, new_y)

    ## @brief Subtracts argument object from current object
    #  @details It assumes the input is of type ComplexT and makes a new
    #           ComplexT that is the subtraction of the argument and current object
    #  @param a ComplexT to subtract from current object
    #  @return The subtraction of argument from current object as a ComplexT
    def sub(self, s):
        new_x = self.x - s.x
        new_y = self.y - s.y
        return ComplexT(new_x, new_y)

    ## @brief Multiplies argument object to current object
    #  @details It assumes the input is of type ComplexT and makes a new
    #           ComplexT that is the multiplication of argument and current object
    #           and the current object
    #  @param m ComplexT to multiply to current object
    #  @return The multiplication of argument and current object as a ComplexT
    def mult(self, m):
        new_x = self.x*m.x - self.y*m.y
        new_y = self.x*m.y + self.y*m.x
        return ComplexT(new_x, new_y)

    ## @brief Gets the reciprocal of the current object
    #  @details It assumes that it makes a new ComplexT that is the reciprocal
    #          of the current object, and input is not z = 0 + 0i
    #  @return The division of current object by argument as a ComplexT
    def recip(self):
        new_x = self.x/(self.x**2 + self.y**2)
        new_y = (-1)*(self.y/(self.x**2 + self.y**2))
        return ComplexT(new_x, new_y)

    ## @brief Divides current object by argument object
    #  @details It assumes the input is of type ComplexT and makes a new
    #           ComplexT that is the division of the current object
    #           by the argument, and input is not z = 0 + 0i
    #  @param d ComplexT to divide current object
    #  @return The division of current object by argument as a ComplexT
    def div(self, d):
        frac = 1/(d.x**2 + d.y**2)
        new_x = frac*((self.x*d.x) + (self.y*d.y))
        new_y = frac*((self.y*d.x) - (self.x*d.y))
        return ComplexT(new_x, new_y)

    ## @brief Gets the positive square root current object
    #  @details It assumes it makes a new ComplexT that is the positive
    #           square root of the current object, and input is not z = 0 + 0i
    #  @return If imaginary part is 0, it returns the square root of the real
    #          part. If not, returns
    def sqrt(self):
        if (self.y == 0):
            return ComplexT(math.sqrt(self.x), self.y)
        sq_ab = math.sqrt(self.x**2 + self.y**2)
        new_x = math.sqrt((self.x + sq_ab)/2)
        sgn = 1
        if self.y < 0:
            sgn = -1
        new_y = sgn*math.sqrt((((-1)*self.x) + sq_ab)/2)
        return ComplexT(new_x, new_y)
```

G Code for triangle_adt.py

```
## @file triangle_adt.py
#  @author Cassidy Baldin
#  @brief Contains a class for creating a triangle
#  @date January 21st, 2021

import math
from enum import Enum

## @brief An ADT for triangles
#  @details A triangle composed of three sides; x, y, and z
class TriangleT:

    ## @brief Constructor for TriangleT
    #  @details Creates a triangle composed of three sides; x, y, and z.
    #           It is assumed that the input sides will be positive,
    #           non-zero integer values.
    #  @param x Integer representing side x
    #  @param y Integer representing side y
    #  @param z Integer representing side z
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

    ## @brief Gets the sides of the triangles
    #  @return Integer tuple value representing sides as a tuple
    def get_sides(self):
        return (self.x, self.y, self.z)

    ## @brief Checks if argument and current object are equal
    #  @details They are considered to be equal if all side lengths are equal.
    #           It assumes the input is of type TriangleT.
    #  @param e TriangleT to compare to current object
    #  @return True if the argument and the current object are equal, else False
    def equal(self, e):
        list1 = [e.x, e.y, e.z]
        list2 = [self.x, self.y, self.z]
        list1.sort()
        list2.sort()
        if (list1[0] == list2[0]) and (list1[1] == list2[1]) and (list1[2] == list2[2]):
            return True
        return False

    ## @brief Gets the perimeter of the current triangle
    #  @return Integer representing the perimeter of the triangle
    def perim(self):
        return self.x + self.y + self.z

    ## @brief Gets the area of the current triangle
    #  @details It assumes input is valid triangle. Uses equation from this
    #           source: https://www.mathsisfun.com/geometry/herons-formula.html
    #  @return Float representing the area of the triangle
    def area(self):
        s = self.perim()/2
        return math.sqrt(s*(s-self.x)*(s-self.y)*(s-self.z))

    ## @brief Checks whether or not the triangle is valid
    #  @details Considered valid if the sum of two sides is smaller than third side.
    #           Uses equation from this source:
    #           https://www.wikihow.com/Determine-if-Three-Side-Lengths-Are-a-Triangle
    #  @return True if valid triangle, else False
    def is_valid(self):
        x = self.x
        y = self.y
        z = self.z
        if (x + y > z) and (x + z > y) and (y + z > x):
            return True
        return False

    ## @brief Returns a TriType corresponding to the type of triangle that was input
    #  @details Creates an element of set {equilat, isosceles, scalene, right}.
    #           The assumed priority label is that if the triangle is a
    #           right triangle, it will take on this identity first. It also
    #           assumes that the triangle is valid.
    #  @return TriType element that corresponds to the type of triangle it is.
    def tri_type(self):
```

```
        r = (self.x**2 + self.y**2 == self.z**2) or (self.x**2 + self.z**2 == self.y**2) or (self.z**2
            + self.y**2 == self.x**2)
        eq = (self.x == self.y == self.z)
        iso = (self.x == self.y) or (self.x == self.z) or (self.y == self.z)
        scal = (self.x != self.y != self.z)

        if (r):
            return TriType(1)
        elif(eq):
            return TriType(2)
        elif(iso):
            return TriType(3)
        return TriType(4)

## @brief A Enum class for triangle types
#  @details Priority order is right, equilat, isosceles, scalene
class TriType(Enum):
            right = 1
            equilat = 2
            isosceles = 3
            scalene = 4
```

H Code for test_driver.py

```
## @file test_driver.py
#  @author Cassidy Baldin
#  @brief Tests for complex_adt.py and triangle_adt.py
#  @date January 21st, 2021

#Citation: some code taken from test_expt.py file

from complex_adt import ComplexT
from triangle_adt import TriangleT, TriType
import math

###TEST CASES FOR COMPLEX_ADT.PY###
complex_test_pass_count = 0
complex_test_fail_count = 0
complex_test_total = 0
a = ComplexT(3.0, 4.0)
b = ComplexT(1.5, -2.0)
c = ComplexT(-2.0, 3.0)
zero_x = ComplexT(0.0, 2.5)
zero_y = ComplexT(2.5, 0.0)

# real
if (a.real() == 3.0):
    complex_test_pass_count += 1
else:
    print("real test FAILS")
    complex_test_fail_count += 1
complex_test_total += 1

# imag
if (a.imag() == 4.0):
    complex_test_pass_count += 1
else:
    print("imag test FAILS")
    complex_test_fail_count += 1
complex_test_total += 1

# get_r
if (a.get_r() == 5.0):
    complex_test_pass_count += 1
else:
    print("r test FAILS")
    complex_test_fail_count += 1
if (b.get_r() == 2.5):
    complex_test_pass_count += 1
else:
    print("r test2 FAILS")
    complex_test_fail_count += 1
complex_test_total += 2

# get_phi
if (math.isclose(a.get_phi(), 0.927295218, abs_tol = 0.00000001)):
    complex_test_pass_count += 1
else:
    print("phi test FAILS")
    complex_test_fail_count += 1
if (math.isclose(b.get_phi(), -0.927295218, abs_tol = 0.00000001)):
    complex_test_pass_count += 1
else:
    print("phi test2 FAILS")
    complex_test_fail_count += 1
if (math.isclose(c.get_phi(), 2.158798931, abs_tol = 0.00000001)):
    complex_test_pass_count += 1
else:
    print("phi test3 FAILS")
    complex_test_fail_count += 1
complex_test_total += 3

# equal
eq_float = ComplexT(3.00000000012345, 4.0)
eq_pre = ComplexT(0.3+0.3+0.3, 0)
if ((a.equal(ComplexT(3.0, 4.0)))):
    complex_test_pass_count += 1
else:
    print("complex equal test FAILS")
    complex_test_fail_count += 1
```

```python
if (a.equal(ComplexT(3.0, -4.0)) == False):
    complex_test_pass_count += 1
else:
    print("complex equal test2 FAILS")
    complex_test_fail_count += 1
# testing zero and float rounding case (should round to true ans)
if (zero_x.equal(ComplexT(0, 2.50000000000000012))):
    complex_test_pass_count += 1
else:
    print("complex equal test3 FAILS")
    complex_test_fail_count += 1
# rounded to 9 decimal places, they should be equal, imag is equal from above
if (math.isclose(a.real(), eq_float.real(), abs_tol = 0.00000001)):
    complex_test_pass_count += 1
else:
    print("complex equal test4 FAILS")
    complex_test_fail_count += 1
# test for float precision error (eq_pre.real() = 0.89999999...)
if (eq_pre.equal(ComplexT(0.9, 0.0))):
    complex_test_pass_count += 1
else:
    print("add/equal test FAILS")
    complex_test_fail_count += 1
complex_test_total += 5

# conj
a_conj = a.conj()
b_conj = b.conj()
if (a_conj.equal(ComplexT(3.0, -4.0))):
    complex_test_pass_count += 1
else:
    print("conj test FAILS")
    complex_test_fail_count += 1
if (b_conj.equal(ComplexT(1.5, 2.0))):
    complex_test_pass_count += 1
else:
    print("conj test2 FAILS")
    complex_test_fail_count += 1
complex_test_total += 2

# add
add_ab = a.add(b)
add_ac = a.add(c)
if (add_ab.equal(ComplexT(4.5, 2.0))):
    complex_test_pass_count += 1
else:
    print("add test FAILS")
    complex_test_fail_count += 1
if (add_ac.equal(ComplexT(1.0, 7.0))):
    complex_test_pass_count += 1
else:
    print("add test2 FAILS")
    complex_test_fail_count += 1
complex_test_total += 2

# sub
sub_ab = a.sub(b)
sub_ac = a.sub(c)
if (sub_ab.equal(ComplexT(1.5, 6.0))):
    complex_test_pass_count += 1
else:
    print("sub test FAILS")
    complex_test_fail_count += 1
if (sub_ac.equal(ComplexT(5.0, 1.0))):
    complex_test_pass_count += 1
else:
    print("sub test2 FAILS")
    complex_test_fail_count += 1
complex_test_total += 2

# mult
mult_ab = a.mult(b)
mult_ac = a.mult(c)
mult_azx = a.mult(zero_x)
mult_azy = a.mult(zero_y)
if (mult_ab.equal(ComplexT(12.5, 0))):
    complex_test_pass_count += 1
else:
    print("mult test FAILS")
    complex_test_fail_count += 1
```

```python
if (mult_ac.equal(ComplexT(-18.0, 1.0))):
    complex_test_pass_count += 1
else:
    print("mult test2 FAILS")
    complex_test_fail_count += 1
if (mult_azx.equal(ComplexT(-10.0, 7.5))):
    complex_test_pass_count += 1
else:
    print("mult test3 FAILS")
    complex_test_fail_count += 1
if (mult_azy.equal(ComplexT(7.5, 10))):
    complex_test_pass_count += 1
else:
    print("mult test4 FAILS")
    complex_test_fail_count += 1
complex_test_total += 4

# recip
recip_a = a.recip()
recip_b = b.recip()
recip_zx = zero_x.recip()
recip_zy = zero_y.recip()
if (recip_a.equal(ComplexT(0.12, -0.16))):
    complex_test_pass_count += 1
else:
    print("recip test FAILS")
    complex_test_fail_count += 1
if (recip_b.equal(ComplexT(0.24, 0.32))):
    complex_test_pass_count += 1
else:
    print("recip test2 FAILS")
    complex_test_fail_count += 1
if (recip_zx.equal(ComplexT(0.0, -0.4))):
    complex_test_pass_count += 1
else:
    print("recip test3 FAILS")
    complex_test_fail_count += 1
if (recip_zy.equal(ComplexT(0.4, 0.0))):
    complex_test_pass_count += 1
else:
    print("recip test4 FAILS")
    complex_test_fail_count += 1
complex_test_total += 4

# div
div_ab = a.div(b)
div_ac = a.div(c)
div_azx = a.div(zero_x)
div_azy = a.div(zero_y)
if (div_ab.equal(ComplexT(-0.56, 1.92))):
    complex_test_pass_count += 1
else:
    print("div test FAILS")
    complex_test_fail_count += 1
if (div_ac.equal(ComplexT(0.4615384615, -1.307692307))):
    complex_test_pass_count += 1
else:
    print("div test2 FAILS")
    complex_test_fail_count += 1
if (div_azx.equal(ComplexT(1.6, -1.2))):
    complex_test_pass_count += 1
else:
    print("div test3 FAILS")
    complex_test_fail_count += 1
if (div_azy.equal(ComplexT(1.2, 1.6))):
    complex_test_pass_count += 1
else:
    print("div test4 FAILS")
    complex_test_fail_count += 1
complex_test_total += 4

# sqrt
sq_a = a.sqrt()
sq_b = b.sqrt()
sq_zx = zero_x.sqrt()
sq_zy = zero_y.sqrt()
if (sq_a.equal(ComplexT(2.0, 1.0))):
    complex_test_pass_count += 1
else:
    print("sqrt test FAILS")
```

```
    complex_test_fail_count += 1
if (sq_b.equal(ComplexT(1.414213562, -0.707106781))):
    complex_test_pass_count += 1
else:
    print("sqrt test2 FAILS")
    complex_test_fail_count += 1
if (sq_zx.equal(ComplexT(1.118033988, 1.118033988))):
    complex_test_pass_count += 1
else:
    print("sqrt test3 FAILS")
    complex_test_fail_count += 1
if (sq_zy.equal(ComplexT(1.581138830, 0))):
    complex_test_pass_count += 1
else:
    print("sqrt test4 FAILS")
    complex_test_fail_count += 1
complex_test_total += 4

###TEST CASES FOR TRIANGLE_ADT.PY###
triangle_test_pass_count = 0
triangle_test_fail_count = 0
triangle_test_total = 0
t1 = TriangleT(3, 4, 5)
t2 = TriangleT(4, 3, 5)
t3 = TriangleT(1, 4, 3)
t4 = TriangleT(3, 4, 3)

# get_sides
if (t1.get_sides() == (3, 4, 5)):
    triangle_test_pass_count += 1
else:
    print("side test FAILS")
    triangle_test_fail_count += 1
triangle_test_total += 1

# equal
if (t1.equal(t2)):
    triangle_test_pass_count += 1
else:
    print("triangle equal test FAILS")
    triangle_test_fail_count += 1
if (t1.equal(t3) == False):
    triangle_test_pass_count += 1
else:
    print("triangle equal test2 FAILS")
    triangle_test_fail_count += 1
if (t1.equal(t4) == False):
   triangle_test_pass_count += 1
else:
    print("triangle equal test3 FAILS")
    triangle_test_fail_count += 1
triangle_test_total += 3

# perim
if (t1.perim() == 12):
    triangle_test_pass_count += 1
else:
    print("perim test FAILS")
    triangle_test_fail_count += 1
triangle_test_total += 1

# area
if (t1.area() == 6.0):
    triangle_test_pass_count += 1
else:
    print("area test FAILS")
    triangle_test_fail_count += 1
if (math.isclose(t4.area(), 4.472135954)):
    triangle_test_pass_count += 1
else:
    print("area test2 FAILS")
    triangle_test_fail_count += 1
triangle_test_total += 2

# is_valid
if (t1.is_valid()):
    triangle_test_pass_count += 1
else:
    print("valid test FAILS")
    triangle_test_fail_count += 1
```

```python
if (t3.is_valid() == False):
    triangle_test_pass_count += 1
else:
    print("valid test2 FAILS")
    triangle_test_fail_count += 1
if (t4.is_valid()):
    triangle_test_pass_count += 1
else:
    print("valid test3 FAILS")
    triangle_test_fail_count += 1
triangle_test_total += 3

# tri_type
if (t1.tri_type() == TriType.right):
    triangle_test_pass_count += 1
else:
    print("tri_type test FAILS")
    triangle_test_fail_count += 1
if (t1.tri_type() != TriType.isosceles):
    triangle_test_pass_count += 1
else:
    print("tri_type test2 FAILS")
    triangle_test_fail_count += 1
if (t4.tri_type() == TriType.isosceles):
    triangle_test_pass_count += 1
else:
    print("tri_type test3 FAILS")
    triangle_test_fail_count += 1
triangle_test_total += 3

#####END Of TESTS#####
print()
print("Complex Tests Summary")
if complex_test_total == complex_test_pass_count:
    print("Congrats! All complex_adt.py tests passed")
print("Passed: ", complex_test_pass_count, " Failed: ", complex_test_fail_count)
print("Score: ", complex_test_pass_count, "/", complex_test_total)

print()
print("Triangle Tests Summary")
if triangle_test_total == triangle_test_pass_count:
    print("Congrats! All triangle_adt.py tests passed")
print("Passed: ", triangle_test_pass_count, " Failed: ", triangle_test_fail_count)
print("Score: ", triangle_test_pass_count, "/",triangle_test_total)
```

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
```

```
#  @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
```