Assignment 1 Solution

Cassidy Baldin, baldic1

January 26, 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

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 positve 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.

3 Results of Testing Partner's Code

4 Critique of Given Design Specification

5 Answers to Questions

(a)

(b) ...

F   Code for complex_adt.py

```python
## @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
```

```python
#   @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)
```

4

G   Code for triangle_adt.py

```python
## @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):
```

```python
        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")
```

```python
        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
```

10

```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

```python
## @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
```

```python
#              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

```python
## @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
```