Assignment 1 Solution

Bhavna Pereira, pereib4

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

The ambiguity of instruction steps allowed for assumptions to be made. Within `Complex_adt` assumptions were made such that complex numbers do consider 0 to be feasible as both an imaginary and/or real value. Similarly, the code also sees negative integers and decimal values as valid complex number values. Within `triangle_adt`, it is assumed that each client input will be positive integers. For this reason, no test case uses side lengths of decimal values or values equal to or less than 0. Exceptions made during the implementation of the codes include `ZeroDivisionError` and `ValueError`. A `ZeroDivisionError` is implemented within `Complex_adt`,in the `div()`method, seeing as dividing by 0 results in an undefined answer. A `ValueError` is raised within the `get_sides()`, `perim()\, \verb`area()—, and `tri_type()` methods to signify an invalid input of side lengths; the methods run if and only if the inputted side lengths form a valid triangle. This ensures that operations do not produce inaccurate results on a triangle that is not valid.

2 Test Cases and Rationale

The test cases implemented operate on the premise such that the exceptions and assumptions described in **Assumptions and Exceptions** are in fact valid. The main cases in `Complex_adt` are tested using two positive integer arguments (`ComplexT(2, 3)`, two zero value arguments (`ComplexT(0, 0)`, and two negative integer arguments (`ComplexT(-1, -5)`

to ensure that complex numbers of any value are considered as valid arguments. Most methods in `Complex_adt` is tested using all three of these inputs. To test the functionality of `div()`, an edge case of `ComplexT(0,0)` is used to ensure that the `ZeroDivisionError` is raised.

`Triangle_adt` is tested using a valid equilateral triangle, a valid scalene triangle, a valid right triangle, a valid isosceles triangle, and an invalid triangle to test the functionality of the `tri_type()` method. The remainder of the `Triangle_adt` test cases each use one valid triangle and one invalid triangle to ensure that the methods work, only if the arguments produce a triangle of valid side lengths.

3 Results of Testing Partner's Code

Upon the attempt of testing the partner code files, an error was raised within 0.21 seconds. It was brought to attention that the partner code operates under different assumptions than my code files do. The partner code makes the assumption that 0 is not a valid input for either the real or imaginary parts of a complex number. In order to run the remainder of the test cases, I was required to comment on the assertion made to ensure that neither x nor y can hold the value of 0.

Since none of my test cases for `Triangle_adt` used values of 0, I did not need to comment out their assertion in their corresponding file. Once this issue was fixed, a total of 22 out of 37 test cases passed.

Of the 15 failed cases, only one case failed due to a difference in calculated values. This test was done on the `get_phi()` method. A reason for this could be a result of my using the cmath library while the partner code manually calculated the value. A few of the other failed test cases were due to the fact that the instances differed, despite having the same values or attributes, namely within the `add()`, `sub()`, `mult()` and `conj()` methods. This could be a result of my code implementing a `__eq__(self, complexT)__` method to ensure that the equal values are displayed as the same instance, while the partner code did not. As mentioned in the **Assumptions and Exceptions** section, a few of my `Triangle_adt` methods ensure that invalid triangle inputs will raise a value error. I implemented test cases to verify this on my own code. Upon evaluation, I recognize that the partner code does not raise these errors in methods where my code will, namely within the `get_sides()`, `area()`, `perim()`, and `tri_type()` methods, and thus failed the test.

4 Critique of Given Design Specification

The design of this assignment was initially intimidating, considering Ive never worked through a virtual machine. I have come to appreciate, however, the ample resources

available to ensure a smooth developmental process. The ambiguity of the instruction steps can be taken as either a good design or a difficult-to-navigate design; it makes room for intuitive assumptions, however it could result in a code that does not satisfy the specifications. In order for a code to be correct, it must satisfy its requirement specifications, which are sometimes not clearly defined within this design. Upon completing **Part 1** of the assignment, a student may have a fair amount of confidence in the performance of their submitted code. However, having made assumptions, they risk not passing some tests that may be used externally and thus risk affecting the usability of the code. Despite this, I do not suggest making changes to the ambiguity of the design. I ultimately enjoyed having freedom to make the assumptions and raise exceptions where I feel necessary, as it allows me to think more rationally as a software engineer, as a good software engineer must be comfortable with different levels of abstraction.

5 Answers to Questions

(a) In the `Complex_adt` class, `real()`, `imag()`, `get_r()`, `get_phi()`, `conj()`, and `sqrt()`, are getters. The remainder of the methods are neither getters, nor setters, as they dont alter to manipulate the current values. They instead use the current values to create new instances of the object. In `Triangle_adt` class, `get_sides()`, `perim()`, `area()`, `is_valid()`, and `tri_type()` are getters, as they only take into consideration the current values. The remainder of the methods, as before, are neither getters, nor setters for the same reasons mentioned previously.

(b) In `Complex_adt`, the real and imaginary parts of the complex numbers can be considered state variables. Defined in the `__init__(self, x, y)` method, they are kept track of for use in all methods. Similarly, in `Triangle_adt`, the three side lengths are defined in the `__init__(self, x, y, z)` method and kept track of for use in all methods within the class, and thus can be classified as state variables.

(c) If in the case the user needs to analyze or predict the outcomes of other methods, such as `add()`, `sub()`, and `div()`, the implementation of less than and greater than can be useful. Although the user can determine whether or not a new object is greater than or less than the current object by using the methods mentioned and analyzing the returned values. It can be argued either way, however it is ultimately not important or necessary to have methods for greater than or less than since other methods can help determine this.

(d) It is possible that the three integers input to the constructor for `TriangleT` will not form a geometrically valid triangle. This can be determined within the `is_valid()` method. A triangle is valid if and only if the sum of two sides is greater than the

value of the third side. This case should be true for all sides. This method returns a boolean value; if the triangle is valid, True will be returned and False will be returned otherwise. It is important to consider the validity of the other methods when inputting a geometrically invalid triangle. For this reason, the methods should also first ensure the triangle is valid before moving forth. If the triangle is invalid, a `ValueError` is raised. In doing so, the user is notified that the triangle is not valid and thus the method will not manipulate it to produce an outcome.

(e) This is a good idea, as the classification of the triangle can be stored within the class. This is useful for when `tri_type()` is not required and does not have to be determined by the user; it is instead given as an input. However, since `tri_type()` is implemented, the user must determine the classification using the method and the type of triangle should not be a state variable.

(f) Performance can be seen as an external quality; it can be judged by the user. It has a direct relationship with usability in that poor performance affects the usability of the software. Despite the content or correctness of the code, an inability to perform makes it unusable. Usability can be determined through the user interface; the user will either know or be unaware of how to use a product without instructions. The concept of usability can be further strengthened using standardization. It is important to provide a universal basis of communication with, or standardize, the software whether it is in the form of syntax rules or knowing how to provide a valid input or command line when prompted.

(g) To fake a rational design process is to assume the correct software is the one that the developer starts with; documentation is made ignoring all mistakes made throughout the process. This is a result of an inefficient waterfall software design. To simply go forth and provide a faked rational design process is to expect that the final product will be attained by each developer on the first attempt. It is especially necessary to document each step of the development process when working amongst other developers through virtual machines. It is beneficial for each colleague to have access to the history of the development to see which implementations do and do not work as the development progresses.

(h) Reliability is determined by if the software product performs that way it is supposed to do; it is designed in order to meet the instructed specifications. Reusability is determined by if the software product or any part of it can be used to create a new product. It is helpful to reuse components of another reliable product to save time or cost, however issues arise when the specifications of the existing and new products do not coincide. This may result in a reliable, yet incorrect new product.

(i) Programming languages are abstractions atop hardware in many forms. Common examples include the ranges in for loops, or by using `copy()` or `sort()`. Through this, the user can write code without having to worry about the way in which the hardware processes and computes various commands. The same applies for mathematical operations such as **, *, +, -, //, and /; by simply typing these commands, the hardware knows how to accurately compute the intended outcome. Programming languages serve as an interface to communicate with the hardware, such that the language uses abstractions while the hardware uses refinement to produce the desired result.

F Code for complex_adt.py

```
## @file complex_adt.py
#  @author Bhavna Pereira
#  @brief Complex_adt method implementation; step 1 of Assignment 1
#  @date 21/01/2021

import cmath

## @brief This Class creates an object of ComplexT
#  @details The class takes in two integers and formats them as complex numbers.
class ComplexT:

## @brief This is a constructor for ComplexT
#  @details This constructor creates a complex number based on the inputted values
#  @param x An integer representing the real value
#  @param y An integer representing the imaginary value
        def __init__(self, x, y):
                self.__x = x
                self.__y = y

        def __str__(self):
                return "({}, {})".format(self.__x, self.__y)

        def __eq__(self, complexT):
                if complexT.real() == self.__x and complexT.imag() == self.__y:
                        return True
                else:
                        return False

## @brief this method returns the real number of a complex number
#  @details in a complex number of format a + bi, this method returns the value a
#  @return the returned value in the form of an integer
        def real(self):
                return self.__x

## @brief this method returns the imaginary number of a complex number
#  @details in a complex number of format a + bi, this method returns the value b
#  @return the returned value in the form of an integer
        def imag(self):
                return self.__y

## @brief this method returns the radius of a complex number
#  @details in a complex number of format a + bi, this method
#  returns the square root of (a^2 + b^2)
#  @return the returned value in the form of a float
        def get_r(self):
                return abs(sqrt(x**2 + y**2))

## @brief this method returns the phase of a complex number
#  @details in a complex number of format a + bi, this method uses the cmath
#  library to compute the phase of the complex number
#  @return the returned value in the form of a float
        def get_phi(self):
                comp = complex(self.__x,self.__y)
                return cmath.phase(comp)

## @brief this method computes if the current complex number
#   equals another inputted complex number
#  @details in a complex number of format a + bi, this method
#  compares the a and b values of each to check if they're equal
#  @param the method takes in another object of type ComplexT
#  @return the returned value in the form of a boolean
        def equal(self, complexT):
                return self.__eq__(complexT)

## @brief this method computes the conjugate of a complex number
#  @details in a complex number of format a + bi, this method
#  computes the negative of the imaginary value, i.e., the negative
#  of value b
#  @return the returned value in the form of an integer object of type ComplexT
        def conj(self):
                return ComplexT(self.__x, -self.__y)

## @brief this method adds to the current complex number
#  another inputted complex number
#  @details in two complex numbers of format a + bi, c + di, this method
#  adds a+c and b+d
```

```
#  @param the method takes in another object of type ComplexT
#  @return the returned value in the form of an integer
        def add(self, complexT):
                return ComplexT(complexT.real() + self.__x, complexT.imag() + self.__y)

## @brief this method subtracts from the current complex number
#  another inputted complex number
#  @details in two complex numbers of format a + bi, c + di, this method
#  subtracts a-c and b-d
#  @param the method takes in another object of type ComplexT
#  @return the returned value in the form of an integer
        def sub(self, complexT):
                return ComplexT(self.__x - complexT.real(), self.__y - complexT.imag())

## @brief this method multiplies the values of the current complex number
#  by the values of another inputted complex number
#  @details in two complex numbers of format a + bi, c + di, this method
#  multiplies a*c and b*d
#  @param the method takes in another object of type ComplexT
#  @return the returned value in the form of an integer
        def mult(self, complexT):
                return ComplexT((self.__x * complexT.real()), (self.__y * complexT.imag()))

## @brief this method calculates the reciprocal of a complex number
#  @details in the form of a + bi, this method calculates the inverse
#  of a + bi
#  @return the returned value in the form of an float
        def recip(self):
                comp = (1/(self.__x + self.__y*1j))
                return ComplexT(comp.real, comp.imag)


## @brief this method divides the values of the current complex number
#  by the values of another inputted complex number
#  @details in two complex numbers of format a + bi, c + di, this method
#  multiplies a/c and b/d. It raises an exception if the inputted values are 0,
#  as a number cannot be divided by 0
#  @param the method takes in another object of type ComplexT
#  @return the returned value in the form of an float
        def div(self, complexT):
                if complexT.real() == 0 or complexT.imag() == 0:
                        raise ZeroDivisionError("Cannot Divide By Zero")
                else:
                        return ComplexT((self.__x / complexT.real()), (self.__y / complexT.imag()))

## @brief this method finds the square root of a complex number
#  @details in the complex number of format a + bi, this method
#  uses the cmath library to determine the square root of a + bi
#  @return the returned value in the form of an integer object of type ComplexT
        def sqrt(self):
                ans = cmath.sqrt(self.__x + self.__y*1j)
                return ComplexT(ans.real, ans.imag)
```

G Code for triangle_adt.py

```
## @file triangle_adt.py
#  @author Bhavna Pereira
#  @brief Triangle_adt method implementation; step 1 of Assignment 1
#  @date 21/01/2021

from math import sqrt
from enum import Enum, auto

## @brief This Class creates an object of type TriType
#  @details The class determine the classification in the form
#  of a triangle, based on their values
class TriType(Enum):
        equilat = auto()
        isosceles = auto()
        scalene = auto()
        right = auto()

## @brief This Class creates an object of type TriangleT
#  @details The class takes in three integers and uses their values as
#  the side lengths of triangles to calculate various functions
class TriangleT:

## @brief This is a constructor for TriangleT
#  @details This constructor creates a triangle based on the inputted values
#  @param x An integer representing the first side length
#  @param y An integer representing the second side length
#  @param z An integer representing the third side length
        def __init__(self, x,y,z):
                self.__x = x
                self.__y = y
                self.__z = z

        def __eq__(self, triangleT):
                inputted = [triangleT.__x, triangleT.__y, triangleT.__z]
                inputted.sort()
                current = [self.__x, self.__y, self.__z]
                current.sort()
                if inputted == current:
                        return True
                else:
                        return False

## @brief this method differentiates the sides of the triangle
#  @details the method checks to see if the triangle is valid, and if it is it returns
#  the side lengths
#  @return the returned values are in the form of integers
        def get_sides(self):
                if self.is_valid() == False:
                        raise ValueError ("This is not a valid triangle")
                else:
                        return (self.__x, self.__y, self.__z)

## @brief this method checks to see if the current side lengths are equal
#  @details the method check equality of the side lenghts of the inputted TriangleT object
#  by passing it to "__eq__"
#  @return the returned value is in the form of boolean
        def equal(self, triangleT):
                return self.__eq__(triangleT)

## @brief this method calculates the perimeter of the TriangleT object
#  @details The method verifies if the triangle is valid, and if it is,
#  it calculates the perimeter by adding together the three side lengths
#  @return the returned value is in the form of an integer
        def perim(self):
                if self.is_valid() == False:
                        raise ValueError ("This is not a valid triangle")
                else:
                        return (self.__x + self.__y + self.__z)

## @brief this method calculates the area of the TriangleT object
#  @details The method verifies if the triangle is valid, and if it is,
#  it calculates the perimeter using heron's formula
#  @return the returned value is in the form of a float
        def area(self):
                if self.is_valid() == False:
                        raise ValueError ("This is not a valid triangle")
```

```
                else:
                        s = self.perim() / 2
                        return (sqrt(s * (s-self.__x) * (s-self.__y) * (s-self.__z)))

## @brief this method verifies the validity of the object of type TriangleT
#  @details The method checks to see if the addition of any two of the three side
#  lengths have a sum greater than the value of the third side length. If this is
#  not true, the triangle is invalid
#  @return the returned value is in the form of a  boolean
        def is_valid(self):
                if self.__x + self.__y > self.__z and self.__y + self.__z > self.__x:
                        if self.__x + self.__z > self.__y:
                                return True
                        else:
                                return False
                else:
                        return False

## @brief this method calculates the type of triangle of the TriangleT object
#  @details The method verifies if the triangle is valid, and if it is,
#  it calculates whether the triangle is scalene, right, isosceles, or equilateral
#  based on a variety of interdependent properties of each side length value
#  @return the returned value is in the form of TriType.x, where x represents
#  either scalene, right, equilateral, or isosceles
        def tri_type(self):
                if self.is_valid() == False:
                        raise ValueError ("This is not a valid triangle")
                elif self.__x == self.__y and self.__y == self.__z:
                        return TriType.equilat
                elif self.__x == self.__y and self.__y != self.__z:
                        return TriType.isosceles
                elif self.__x == self.__z and self.__z != self.__y:
                        return TriType.isosceles
                elif self.__y == self.__z and self.__z != self.__x:
                        return TriType.isosceles
                elif self.__y != self.__x and self.__x != self.__z and self.__y != self.__z:
                        if ((self.__x)**2 + (self.__y**2)) == (self.__z)**2:
                                return TriType.right
                        if ((self.__x)**2 + (self.__z)**2) == (self.__y)**2:
                                return  TriType.right
                        if ((self.__y)**2 + (self.__z)**2) == (self.__x)**2:
                                return TriType.right
                        else:
                                return TriType.scalene
```

H Code for test_driver.py

```
## @file test_driver.py
#  @author Bhavna Pereira
#  @brief The following aim to test the validity of the following files: triangle_adt, complex_T
#  @date 20/01/2021


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


validInput1 = ComplexT(2, 3)
compEdgeCase1 = ComplexT(0, 0)
compEdgeCase2 = ComplexT(-1, -5)
comp = ComplexT(2, 3)
comp2 = ComplexT(3, 3)
comp3 = ComplexT(10, 4)
comp4 = ComplexT(3, 4)

validIsosceles = TriangleT(3,2,2)
validScalene = TriangleT(4,2,3)
validRight = TriangleT(3,4,5)
validEquilat = TriangleT(3,3,3)
triEdgeCase1 = TriangleT(0, 1, 2)
triEdgeCase2 = TriangleT(0.5, 1, 1.5)
triEdgeCase2 = TriangleT(-3, 4, 5)
tri = TriangleT(3, 3, 3)
tri2 = TriangleT(4, 2, 7)

def test_Real():
        expectedReal = 2
        assert validInput1.real() == expectedReal

def test_edge_Real():
        expectedReal1 = 0
        assert compEdgeCase1.real() == expectedReal1

def test_edge2_Real():
        expectedReal2 = -1
        assert compEdgeCase2.real() == expectedReal2


def test_Imag():
        expectedImag = 3
        assert validInput1.imag() == expectedImag


def test_edge_Imag():
        expectedImag1 = 0
        assert compEdgeCase1.imag() == expectedImag1


def test_edge2_Imag():
        expectedImag2 = -5
        assert compEdgeCase2.imag() == expectedImag2


def r_test():
        expectedR = 3.6055
        assert pytest.approx(validInput1.get_r(), rel=1e-3) == expectedR


def r_edge_test():
        expectedR1 = 0
        assert pytest.approx(compEdgeCase1.get_r(), rel=1e-3) == expectedR1

def r_edge2_test():
        expectedR2 = 5.0990
        assert pytest.approx(compEdgeCase2.get_r(), rel=1e-3) == expectedR2


def test_getPhi():
        expectedPhi = 0.9828
        assert pytest.approx(validInput1.get_phi(), rel=1e-3) == expectedPhi
```

```
def test_edge_getPhi():
        expectedPhi1 = 0
        assert pytest.approx(compEdgeCase1.get_phi(), rel=1e-3) == expectedPhi1


def test_edge2_getPhi():
        expectedPhi2 = -1.7682
        assert pytest.approx(compEdgeCase2.get_phi(), rel=1e-3) == expectedPhi2


def test_equal():
        assert validInput1.equal(comp) == True


def test2_equal():
        assert validInput1.equal(comp3) == False


def test_conj():
        expectedConj = ComplexT(2,-3)
        assert validInput1.conj() == expectedConj


def test_edge_conj():
        expectedConj1 = ComplexT(-1, 5)
        assert compEdgeCase2.conj() == expectedConj1


def test_add():
        expectedAdd = ComplexT(4, 6)
        assert validInput1.add(comp) == expectedAdd


def test_edge2_add():
        expectedAdd1 = ComplexT(1, -2)
        assert compEdgeCase2.add(comp) == expectedAdd1


def test_sub():
        expectedSub = ComplexT(0, 0)
        assert validInput1.sub(comp) == expectedSub


def test_edge2_sub():
        expectedSub1 = ComplexT(3, 8)
        assert comp.sub(compEdgeCase2) == expectedSub1


def test_mult():
        expectedMult = ComplexT(4, 9)
        assert validInput1.mult(comp) == expectedMult


def test_recip_real():
        expectedRecip = ComplexT(0.1538, -0.2308)
        assert pytest.approx(validInput1.recip().real(), rel=1e-3) == expectedRecip.real()


def test_recip_imag():
        expectedRecip = ComplexT(0.1538, -0.2308)
        assert pytest.approx(validInput1.recip().imag(), rel=1e-3) == expectedRecip.imag()


def test_div():
        expectedDiv = ComplexT(1, 1)
        assert validInput1.div(comp) == expectedDiv


def test_sqrt():
        expectedsqrt = ComplexT(2, 1)
        assert comp4.sqrt() == expectedsqrt

def test_edge_div():
        with pytest.raises(ZeroDivisionError) as e:
                validInput1.div(compEdgeCase1)
        assert "Cannot Divide By Zero" == str(e.value)


def test_get_sides():
        expectedSides = (3, 2, 2)
```

```
        assert validIsosceles.get_sides() == expectedSides


def test_edge_get_sides():
        with pytest.raises(ValueError) as e:
                triEdgeCase1.get_sides()
        assert "This is not a valid triangle" == str(e.value)


def test_equal():
        excpectedEqual = True
        assert validEquilat.equal(tri) == excpectedEqual


def testtri_equal():
        expectedEqual2 = False
        assert validIsosceles.equal(tri) == expectedEqual2


def test_perim():
        expectedPerim = 12
        assert validRight.perim() == expectedPerim


def test2_perim():
        with pytest.raises(ValueError) as e:
                triEdgeCase1.perim()
        assert "This is not a valid triangle" == str(e.value)


def test_area():
        expectedArea = 1.9843
        assert pytest.approx(validIsosceles.area(), rel=1e-3) == expectedArea


def test_edge_area():
        with pytest.raises(ValueError) as e:
                triEdgeCase1.area()
        assert "This is not a valid triangle" == str(e.value)


def test_isValid():
        expectedValid = True
        assert validEquilat.is_valid() == expectedValid


def test2_isValid():
        expectedValid2 = False
        assert triEdgeCase1.is_valid() == expectedValid2


def test1_tritype():
        expectedType = TriType.isosceles
        assert validIsosceles.tri_type() == expectedType


def test2_tritype():
        expectedType2 = TriType.equilat
        assert validEquilat.tri_type() == expectedType2


def test3_tritype():
        expectedType3 = TriType.right
        assert validRight.tri_type() == expectedType3


def test4_tritype():
        expectedType4 = TriType.scalene
        assert validScalene.tri_type() == expectedType4


def test5_tritype():
        with pytest.raises(ValueError) as e:
                triEdgeCase1.tri_type()
        assert "This is not a valid triangle" == str(e.value)
```

I Code for Partner's complex_adt.py

```
## @file complex_adt.py
#  @brief Contains the ComplexT class which represents complex numbers
#  @author Nathan Uy
#  @date 01/21/2021

from math import sqrt, atan, atan2, pi

## @brief An abstract data type that represents complex numbers.
class ComplexT:

        ## @brief A ComplexT constructor.
        # @details Initializes a ComplexT object with the real and imaginary part.
        # @param x A float that represents the real part of the complex number.
        # @param y A float that represents the imaginary part of the complex number.
        # @throws AssrtionError Throws AssertionError if x and y are equal to zero.
        def __init__(self, x, y):
                ##assert (not(x == 0 and y == 0)), "x and y must be non-zero."
                self.x = x
                self.y = y

        ## @brief Gets the real part of the complex number.
        # @return A float that represents the real part of the complex number.
        def real(self):
                return self.x

        ## @brief Gets the imaginary part of the complex number.
        # @return A float that represents the imaginary part of the complex number.
        def imag(self):
                return self.y

        ## @brief Calculates the modulus of the complex number.
        # @return A float that represents the modulus of the complex number.
        def get_r(self):
                return sqrt(self.x**2 + self.y**2)

        ## @brief Calculates the phase of the complex number in radians.
        # @details Assuming phi is between 0 and 2pi.
        # @return A float that represents the phase of the complex number.
        def get_phi(self):
                if self.x < 0 and self.y < 0:
                        return pi + atan2(self.y, self.x)
                elif self.x > 0 and self.y < 0:
                        return pi + atan2(self.y, self.x)
                else:
                        return atan2(self.y, self.x)

        ## @brief Determines whether the current object and the argument are equal.
        # @param Complex A ComplexT object.
        # @return True if the current object is equal to Complex; False otherwise.
        def equal(self, Complex):
                return (self.x == Complex.x and self.y == Complex.y)

        ## @brief Calculates the complex conjugate of the current object.
        # @return A ComplexT object which represents the complex conjugate of the current object.
        def conj(self):
                y = -1 * self.y
                return ComplexT(self.x, y)

        ## @brief Calculates the sum of the current object and the argument.
        # @param Complex A ComplexT object.
        # @return A ComplexT object which represents the sum of the argument and the current object.
        def add(self, Complex):
                x = self.x + Complex.x
                y = self.y + Complex.y
                return ComplexT(x,y)

        ## @brief Subtracts the argument from the current object.
        # @param Complex A ComplexT object.
        # @return A ComplexT object that subtracts the argument from the current argument.
        def sub(self, Complex):
                x = self.x - Complex.x
                y = self.y - Complex.y
                return ComplexT(x,y)

        ## @brief Calculates the product of the current object and the argument.
        # @param Complex A ComplexT object.
```

```
        # @return A ComplexT object that represents the product of the current object and the argument.
        def mult(self, Complex):
                y = self.x*Complex.y + self.y*Complex.x
                x = self.x*Complex.x - self.y*Complex.y
                return ComplexT(x,y)

        ## @brief Calculates the reciprocal of the current object.
        # @return A ComplexT object that represents the reciprocal of the current object.
        def recip(self):
                divisor = self.x**2 + self.y**2
                x = self.x / divisor
                y = -1 * self.y / divisor
                return ComplexT(x,y)

        ## @brief Divides the current object by the argument.
        # @param Complex A ComplexT object.
        # @return A ComplexT object that divides the current argument by the argument.
        def div(self, Complex):
                x = (self.x*Complex.x + self.y*Complex.y) / (Complex.x**2 + Complex.y**2)
                y = (self.y*Complex.x - self.x*Complex.y) / (Complex.x**2 + Complex.y**2)
                return ComplexT(x,y)

        ## @brief Calculates the squareroot of the current object.
        # @return A ComplexT object that represents the squareroot of the current object
        # @throws ZeroDivisionError Throws ZeroDivisionError if y is equal to zero.
        # @throws ValueError Throws ValueError if the modulo of the object + x < 0 or the modulo of
            the object - x < 0).
        def sqrt(self):
                if self.y == 0:
                        raise ZeroDivisionError("The imaginary part of the complex number must be
                            non-zero for sqrt to exist.")
                elif (self.get_r() + self.x < 0 or self.get_r() - self.x < 0):
                        raise ValueError("This complex number does not have a squareroot")
                else:
                        x = sqrt((self.get_r() + self.x) / 2)
                        y = (self.y / abs(self.y)) * sqrt((self.get_r() - self.x) / 2)
                return ComplexT(x,y)
```

J Code for Partner's triangle_adt.py

```
## @file triangle_adt.py
#  @brief Contains the TriangleT class which represents triangles
#  @author Nathan Uy
#  @date 01/21/2021

from math import sqrt
from enum import Enum, auto

## @brief An abstract data type that represents triangles.
class TriangleT:

        ## @brief TriangleT constructor
        # @details Initializes a TriangleT object with the three sides.
        # @param l1 An integer that represents the length of one side of the triangle.
        # @param l2 An integer that represents the length of one side of the triangle.
        # @param l3 An integer that represents the length of one side of the triangle.
        # @throws AssertionError Throws AssertionError if one of the sides is equal to zero.
        def __init__(self, l1, l2, l3):
                ##assert(l1 > 0 and l2 > 0 and l3 > 0), "All three sides must be greater than 0"
                self.l1 = l1
                self.l2 = l2
                self.l3 = l3

        ## @brief Gets the length of the three sides of the triangle.
        # @return A tuple with the lengths of each side of the triangle.
        def get_sides(self):
                return (self.l1, self.l2, self.l3)

        ## @brief Determines whether the current object and the argument are equal.
        # @param Triangle A TriangleT object.
        # @return True if the current object and the argument are equal; False otherwise.
        def equal(self, Triangle):
                currObj = list(self.get_sides())
                arg = list(Triangle.get_sides())
```

```
            currObj.sort()
            arg.sort()
            compare = currObj == arg
            return compare

    ## @brief Calculates the perimeter of the current object.
    ## @details It is assumed that the given triangle is valid
    # @return An integer that represents the perimeter of the current object.
    def perim(self):
            return self.l1+self.l2+self.l3

    ## @brief Calculates the area of the current object.
    ## @details It is assumed that the given triangle is valid
    # @return A float that represents the area of the current object.
    # @throws AssertionError Throws AssertionError if half the sum of the sides is less than or
        equal the greatest side.
    def area(self):
            s = (self.l1+self.l2+self.l3)/2
            assert(s > max(self.get_sides()))
            area = sqrt(s*(s-self.l1)*(s-self.l2)*(s-self.l3))
            return area

    ## @brief Determines whether the three sides of the current object form a valid triangle or
        not.
    # @return True if the three sides of the current object form a valid triangle; False otherwise.
    def is_valid(self):
            if (self.l1+self.l2 > self.l3 and self.l1+self.l3 > self.l2 and self.l3+self.l2 >
                self.l1):
                    return True
            else:
                    return False

    ## @brief Determines the triangle type of the current object.
    ## @details It is assumed that the given triangle is valid
    # @return A TriType object that corresponds to the triangle type of the current object.
    def tri_type(self):
            sides = list(self.get_sides())
            AandB = list(self.get_sides())
            hypotenuse = max(sides)
            AandB.remove(hypotenuse)

            if AandB[0]**2 + AandB[1]**2 == hypotenuse**2:
                    triangleType = TriType(1)
            elif sides[0] == sides[1] and sides[0] == sides[2]:
                    triangleType = TriType(2)
            elif sides[0] == sides[1] or sides[0] == sides[2] or sides[2] == sides[1]:
                    triangleType = TriType(3)
            else:
                    triangleType = TriType(4)

            return triangleType


## @brief TriType constructor.
# @details Creates an enumeration that corresponds to the triangle types.
# @param Enum An integer that represents the type of the triangle.
class TriType(Enum):
        right = auto()
        equilat = auto()
        isosceles = auto()
        scalene = auto()
```