Assignment 1 Solution

By Safwan Hossain, hossam18

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

Assumption: Assume the imaginary part of complex number isnt zero. Some methods require the complex number to be valid due to complications such as zero division.

Assumption: Assume the current triangle is valid. Methods such as tri_type() and area() required this assumption because if the triangle was invalid, the area and triangle type would be undefined.

2 Test Cases and Rationale

For simple methods such as finding the magnitude of complex numbers, it was enough to use random complex numbers due to straightforward calculations. But for methods with special cases like equivalence for triangles, the test case would have to account for the fact that it works with a triangle with the same side lengths even if the lengths are assigned to different variables.

3 Results of Testing Partner's Code

My partner and I passed every test case. Note that I added float point inaccuracy for ComplexT equal() later on.

4 Critique of Given Design Specification

I liked how the design was very simple and straightforward to implement by using general formulas to calculate values for most methods. But I feel like some details were missing such as is a complex number still valid if the imaginary part is 0. One thing I would change is how the triangle types are implemented; scalene, isosceles, and equilateral are all types of triangles based on side length, while a right triangle is based on a triangles angle. Instead it would be better to either just keep the triangle type in terms of side length or introduce the remaining triangle types (acute and obtuse) and return an array of triangle types in following format: [Type based on sides, Type based on angles].

5 Answers to Questions

(a) The methods ComplexT, real(), imag(), get_r(), get_phi() are selectors. The methods TriangleT get_sides(), perim(), area(), tri_type() are selectors. ComplexT doesn't have any mutators because there are no methods that change the parameters of ComplexT, all methods either get or manipulate values without affecting the current object's values. The same applies for TriangleT.

(b) Assuming the classes have dynamic methods, ComplexT can a have a state variable called isComplex (boolean) where it keeps track of the imaginary part to see if it exists, if the imaginary part is zero the variable is false and the object is just a real number which arguably doesn't make it complex. ComplexT can have a state variable called isZero (boolean) where if both parts of the complex number is zero, the variables would be true. TriangleT can have a state variable for keeping track of the type of triangle it is and another state variable (boolean) to keep track if the current triangle is valid (true if valid, false if not valid).

(c) It would not make sense for ComplexT to have greater/less than methods because imaginary numbers cannot be compared in those terms. Imaginary numbers are not comparable because an imaginary number is neither equal, greater nor less than 0, thus it cannot be compared with other imaginary numbers. It is possible to compare other values such as magnitudes, but not the complex numbers themselves.

(d) It is possible for the constructor for TriangleT to form a geometrically non valid triangle. In this case, There are two possible options, which is to let the triangle be constructed or to print out an error message and ask to try again. The problem with the latter solution is that Python will throw an error and stop the program if the triangle is ever called, so the solution would be situational. If it is used in an interface with a user for example, then the console can print out an error message and say the

triangle is not valid and as the user to try again. If however, the constructor is called using premade code (another module) then the triangle should be created (with a warning message), but for methods where geometric validity is required the is_valid() method should be used to check if the triangle is valid. For example If the area() method is called for a non-valid triangle, there should be a message or an impossible value such as -1 being passed so that the given area isnt assumed to be correct.

(e) It is not a practical idea for this assignment to have a state variable for the type of triangle because TriangleT is a static implantation and the triangle type is set upon construction. If the triangle were to be constantly changing and multiple methods are asking for the triangle type, then it would be a good idea for it to have a state variable for its type.

(f) An increase in performance creates a positive impact on usability. This is because the usability of a program will require a certain speed for a program to function. If a program is too slow it will negatively impact usability.

(g) A fake rational design process is very useful when dealing with complex projects, but if a project is very simple and straightforward, it may not be necessary to fake a rational design (although it may be practical).

(h) Reusability can affect reliability in a negative way because it can promote mistakes. Reusable designs can be very general, it is possible for the design to not pay attention too much on every application it is used in. Thus, if a specific application requires you to follow specific safety features or instructions it may not be accounted for in a general design.

(i) Programming languages are an abstraction on hardware because they provide you many ways to use and manipulate the hardware while making it easier to write programs without having to worry about how the hardware is used. The programming language will handle many unnecessary tasks with the hardware for you.

F Code for complex_adt.py

```
## @file complex_adt.py
#  @author Safwan Hossain
#  @brief ADT for complex number
#  @date 1/21/2021

import math


## @brief An ADT for representing complex numbers
#  @details A complex number is defined by the addition of a real number with an
#           imaginary number
class ComplexT:

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

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

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

    ## @brief Gets the magnitude of the complex number
    #  @details Assume the imaginary part of the complex number isn't zero
    #  @return Float representing magnitude of the complex number
    def get_r(self):
        return math.sqrt(self.x ** 2 + self.y ** 2)

    ## @brief Gets the phase of the complex number in radians
    #  @details Assume the imaginary part of the complex number isn't zero
    #  @return Float representing the phase of the complex number in radians
    def get_phi(self):
        return math.atan2(self.y, self.x)

    ## @brief Checks if a given complex number is equivalent to the current
    #         complex number
    #  @param other A complex number
    #  @return True if given complex number is equal to current complex number,
    #          False otherwise
    def equal(self, other):
        maxTol = 10e-9
        xSame = abs(self.x - other.x) <= maxTol
        ySame = abs(self.y - other.y) <= maxTol
        return xSame and ySame

    ## @brief Gets the conjugate of the complex number
    #  @details Assume the imaginary part of the complex number isn't zero
    #  @return ComplexT object that is the conjugate of the current complex number
    def conj(self):
        return ComplexT(self.x, -self.y)

    ## @brief Returns a complex number that is the addition of two complex numbers
    #  @param other A complex number
    #  @return ComplexT object that is the addition of the current complex number and a
    #          given complex number
    def add(self, other):
        return ComplexT(self.x + other.x, self.y + other.y)


    ## @brief Returns a complex number that is the subtraction of two complex numbers
    #  @param other A complex number
    #  @return ComplexT object that is the subtraction of the current complex number from a
    #          given complex number
    def sub(self, other):
        return ComplexT(self.x - other.x, self.y - other.y)
```

```
## @brief Returns a complex number that is the product of two complex numbers
#  @param other A complex number
#  @return ComplexT object that is the product of the current complex number and a
#          given complex number
def mult(self, other):
    x = self.x * other.x - self.y * other.y
    y = self.x * other.y + self.y * other.x

    return ComplexT(x, y)

## @brief Returns the reciprocal of the current complex number
#  @details Assume the imaginary part of the complex number isn't zero
#  @return ComplexT object that is the reciprocal of the current complex number
def recip(self):
    denom = self.x ** 2 + self.y ** 2

    return ComplexT(self.x / denom, -self.y / denom)

## @brief Returns a complex number that is the quotient of two complex numbers
#  @details Assume the imaginary parts of the complex numbers aren't zero
#  @param other A complex number
#  @return ComplexT object that is the quotient of the current complex number and a
#          given complex number
def div(self, other):
    denom = other.x ** 2 + other.y ** 2
    x = (self.x * other.x + self.y * other.y) / denom
    y = (self.y * other.x - self.x * other.y) / denom

    return ComplexT(x, y)

## @brief Returns the square root of the current complex number
#  @details Assume the imaginary part of the complex number isn't zero
#  @return ComplexT object that is the square root of the current complex number
def sqrt(self):
    x = math.sqrt((self.get_r() + self.x) / 2)
    y = (self.y / abs(self.y)) * math.sqrt((self.get_r() - self.x) / 2)

    return ComplexT(x, y)
```

G Code for triangle_adt.py

```
## @file triangle_adt.py
#  @author Safwan Hossain
#  @brief ADT for triangles
#  @date 1/21/2021

import math
from enum import Enum

## @brief An enumerated data type for representing types of triangles
#  @details A triangle can be an equilateral, isosceles, scalene or right triangle
class TriType(Enum):
    equilat = 1
    isosceles = 2
    scalene = 3
    right = 4

## @brief An ADT for representing triangles
#  @details A triangle is composed of 3 connecting sides
class TriangleT:

    ## @brief Constructor for TriangleT
    #  @details Creates a triangle given 3 sides
    #  @param a Integer representing the first side
    #  @param b Integer representing the second side
    #  @param c Integer representing the third side
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    ## @brief Gets all the sides of the current triangle
    #  @return Tuple containing three Integers, each representing a side of
    #          the current triangle
    def get_sides(self):
        return self.a, self.b, self.c

    ## @brief Checks if the current triangle is equivalent to a given triangle
    #  @param other A triangle
    #  @return True if the current triangle is equivalent to a given triangle,
    #          False otherwise
    def equal(self, other):
        return sorted([self.a, self.b, self.c]) == sorted([other.a, other.b, other.c])

    ## @brief Calculates the perimeter of the current triangle
    #  @return Integer representing the perimeter of the current triangle
    def perim(self):
        return self.a + self.b + self.c

    ## @brief Calculates the area of the current triangle
    #  @details Assume the current triangle is valid
    #  @return Float representing the area of the current triangle
    def area(self):
        p = self.perim() / 2
        return math.sqrt(p * (p - self.a) * (p - self.b) * (p - self.c))

    ## @brief Checks if the current triangle is possible with its sides
    #  @return True if triangle is valid, False otherwise
    def is_valid(self):
        a = self.a
        b = self.b
        c = self.c

        return a + b > c and a + c > b and b + c > a

    ## @brief Determines the type of the current triangle
    #  @details Assume the current triangle is valid
    #  @return TriType that represents the type of the current triangle
    def tri_type(self):
        a = self.a
        b = self.b
        c = self.c

        hyp = max(a, b, c)

        if a == b and a == c:
            return TriType.equilat
```

```
elif (a ** 2 + b ** 2 + c ** 2) == 2 * hyp ** 2:
    return TriType.right
elif (a == b and not a == c) or (a == c and not a == b) or (b == c and not a == b):
    return TriType.isosceles
else:
    return TriType.scalene
```

H Code for test_driver.py

```
## @file test_driver.py
#  @author Safwan Hossain
#  @brief test driver for modules ComplexT and TriangleT
#  @date 1/21/2021

import math

from complex_adt import ComplexT
from triangle_adt import TriangleT, TriType


# ComplexT Interface Syntax
def test_check(name, calculatedOutput, expectedOutput):
    counter = 0
    numOfTests = len(calculatedOutput)
    for i in range(numOfTests):
        calculated = calculatedOutput[i]
        expected = expectedOutput[i]
        if calculated == expected:
            counter = counter + 1

    if counter != numOfTests:
        print(name + " test FAILED: Passed " + str(counter) + " / " + str(numOfTests) + " tests")
    else:
        print(name + " test PASSED: Passed " + str(counter) + " / " + str(numOfTests) + " tests")


# ComplexT Interface Syntax
def object_test_check(name, calculatedOutput, expectedOutput):
    counter = 0
    numOfTests = len(calculatedOutput)
    for i in range(numOfTests):
        calculated = calculatedOutput[i]
        expected = expectedOutput[i]
        if calculated.equal(expected):
            counter = counter + 1

    if counter != numOfTests:
        print(name + " test FAILED: Passed " + str(counter) + " / " + str(numOfTests) + " tests")
    else:
        print(name + " test PASSED: Passed " + str(counter) + " / " + str(numOfTests) + " tests")


a = ComplexT(1.0, 2.0)
b = ComplexT(0.5, -0.5)
c = ComplexT(10.0, 5.0)
aCopy = ComplexT(1.0, 2.0)
bCopy = ComplexT(0.5, -0.5)


def real_check():
    test_check("ComplexT real", [a.real(), b.real()], [1.0, 0.5])


def imag_check():
    test_check("ComplexT imag", [a.imag(), b.imag()], [2.0, -0.5])


def get_r_check():
    calculatedOutput = [a.get_r(), b.get_r()]
    expectedOutput = [math.sqrt(1.0 ** 2 + 2.0 ** 2), math.sqrt(0.5 ** 2 + (-0.5) ** 2)]
    test_check("ComplexT get_r", calculatedOutput, expectedOutput)


def get_phi_check():
    calculatedOutput = [a.get_phi(), b.get_phi()]
    expectedOutput = [math.atan2(2.0, 1.0), math.atan2(-0.5, 0.5)]
    test_check("ComplexT get_phi", calculatedOutput, expectedOutput)


def equal_check():
    calculatedOutput = [a.equal(aCopy), b.equal(bCopy), a.equal(b), b.equal(aCopy)]
    expectedOutput = [True, True, False, False]
    test_check("ComplexT equal", calculatedOutput, expectedOutput)
```

```
def conj_check():
    calculatedOutput = [a.conj(), b.conj()]
    expectedOutput = [ComplexT(1.0, -2.0), ComplexT(0.5, 0.5)]
    object_test_check("ComplexT conj", calculatedOutput, expectedOutput)


def add_check():
    calculatedOutput = [a.add(b), a.add(a), b.add(a)]
    expectedOutput = [ComplexT(1.5, 1.5), ComplexT(2, 4), ComplexT(1.5, 1.5)]
    object_test_check("ComplexT add", calculatedOutput, expectedOutput)


def sub_check():
    calculatedOutput = [a.sub(b), c.sub(b), c.sub(a)]
    expectedOutput = [ComplexT(0.5, 2.5), ComplexT(9.5, 5.5), ComplexT(9, 3)]
    object_test_check("ComplexT sub", calculatedOutput, expectedOutput)


def mult_check():
    calculatedOutput = [a.mult(b), a.mult(c), b.mult(a)]
    expectedOutput = [ComplexT(1.5, 0.5), ComplexT(0, 25), ComplexT(1.5, 0.5)]
    object_test_check("ComplexT mult", calculatedOutput, expectedOutput)


def recip_check():
    calculatedOutput = [a.recip(), b.recip()]
    expectedOutput = [ComplexT(1.0 / (1.0 ** 2 + 2.0 ** 2), - 2.0 / (1.0 ** 2 + 2.0 ** 2)),
                      ComplexT(0.5 / (0.5 ** 2 + 0.5 ** 2), 0.5 / (0.5 ** 2 + 0.5 ** 2))]
    object_test_check("ComplexT recip", calculatedOutput, expectedOutput)


def div_check():
    calculatedOutput = [a.div(b), b.div(c)]
    x1 = (1.0 * 0.5 + 2.0 * (-0.5)) / (0.5 ** 2 + 0.5 ** 2)
    y1 = (2.0 * 0.5 - 1.0 * (-0.5)) / (0.5 ** 2 + 0.5 ** 2)
    x2 = (0.5 * 10.0 + (-0.5) * 5) / (10.0 ** 2 + 5.0 ** 2)
    y2 = (-0.5 * 10.0 - 0.5 * 5) / (10.0 ** 2 + 5.0 ** 2)
    expectedOutput = [ComplexT(x1, y1),
                      ComplexT(x2, y2)]
    object_test_check("ComplexT div", calculatedOutput, expectedOutput)


def sqrt_check():
    calculatedOutput = [a.sqrt(), b.sqrt()]
    x1 = math.sqrt((a.get_r() + a.real()) / 2)
    y1 = math.sqrt((a.get_r() - a.real()) / 2) * (a.imag() / abs(a.imag()))
    x2 = math.sqrt((b.get_r() + b.real()) / 2)
    y2 = math.sqrt((b.get_r() - b.real()) / 2) * (b.imag() / abs(b.imag()))
    expectedOutput = [ComplexT(x1, y1),
                      ComplexT(x2, y2)]
    object_test_check("ComplexT sqrt", calculatedOutput, expectedOutput)


# a = ComplexT(1.0, 2.0)
# b = ComplexT(0.5, -0.5)
# c = ComplexT(10.0, 5.0)

real_check()
imag_check()
get_r_check()
get_phi_check()
equal_check()
conj_check()
add_check()
sub_check()
mult_check()
recip_check()
div_check()
sqrt_check()

# TriangleT
t1 = TriangleT(3, 4, 5)
t2 = TriangleT(4, 3, 5)

t1Copy = TriangleT(3, 4, 5)
t3 = TriangleT(43, 32, 51)
t4 = TriangleT(24, 30, 18)
t5 = TriangleT(1, 10, 130)
t6 = TriangleT(1, 1, 1)
```

```
def get_sides_check():
    calculatedOutput = [t1.get_sides(), t2.get_sides()]
    expectedOutput = [(3, 4, 5), (4, 3, 5)]
    test_check("TriangleT get_sides", calculatedOutput, expectedOutput)


def triangle_equal_check():
    calculatedOutput = [t1.equal(t1Copy), t1.equal(t2), t2.equal(t3)]
    expectedOutput = [True, True, False]
    test_check("TriangleT equal", calculatedOutput, expectedOutput)


def perim_check():
    calculatedOutput = [t1.perim(), t2.perim(), t3.perim()]
    expectedOutput = [12, 12, 126]
    test_check("TriangleT perim", calculatedOutput, expectedOutput)


def area_check():
    calculatedOutput = [t1.area(), t2.area(), t4.area()]
    expectedOutput = [6.0, 6.0, 216.0]
    test_check("TriangleT area", calculatedOutput, expectedOutput)


def is_valid_check():
    calculatedOutput = [t1.is_valid(), t2.is_valid(), t5.is_valid()]
    expectedOutput = [True, True, False]
    test_check("TriangleT is_valid", calculatedOutput, expectedOutput)


def tri_type_check():
    calculatedOutput = [t1.tri_type(), t2.tri_type(), t3.tri_type(), t6.tri_type()]
    expectedOutput = [TriType.right, TriType.right, TriType.scalene, TriType.equilat]
    test_check("TriangleT is_valid", calculatedOutput, expectedOutput)


get_sides_check()
triangle_equal_check()
perim_check()
area_check()
is_valid_check()
tri_type_check()
```

I Code for Partner's complex_adt.py

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

import cmath

## @brief An ADT for handling complex numbers
#  @details A complex number contains a real and imaginary part
class ComplexT:

    ## @brief Constructor for ComplexT
    #  @details Creates a complex number based on
    #  a given real value and imaginary value
    #  @details The complex number is x + y*i
    #  @details It is assumed that the passed real and
    #  imaginary parts are Float values
    #  @param x Float representing the real part of the complex number
    #  @param y Float representing the imaginary part of the complex number
    def __init__(self, x, y):
        self.x = x
        self.y = y

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

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

    ## @brief Gets the magnitude of the complex number
    #  @return Float representing the magnitude of the complex number
    def get_r(self):
        return abs(complex(self.x, self.y))

    ## @brief Gets the argument (phase) of the complex number
    #  @details The phase of the complex number, 0 + 0i, is returned as
    #  0 rather than undefined
    #  @details Phase definition obtained from
    #  https://en.wikipedia.org/wiki/Complex_number
    #  @return Float representing the phase in radians
    def get_phi(self):
        return cmath.phase(complex(self.x, self.y))

    ## @brief Checks if another complex number is equal to
    #  the current complex number
    #  @details Uses the __eq__() method to determine approximate equality
    #  @details Approximate equality is chosen
    #  arbitrarily to within 10e-9
    #  @details It is assumed that the input parameter, c,
    #  is a ComplexT object
    #  @param c A complex number (ComplexT object)
    #  @return True if the complex numbers are approximately equal,
    #  False otherwise
    def equal(self, c):
        # Uses the '__eq__()' behaviour to approximately compare ComplexT objects
        return self == c

    # Private method for defining ComplexT '==' behavior
    def __eq__(self, c):
        # Used to avoid floating point errors when comparing complex numbers
        return abs(c.real()-self.x) < 10e-9 and abs(c.imag()-self.y) < 10e-9

    ## @brief Gets the conjugate of the complex number
    #  @return ComplexT object representing the conjugate
    def conj(self):
        return ComplexT(self.x, -1 * self.y)

    ## @brief Adds another complex number to the current complex number
    #  @details It is assumed that the input parameter, c,
    #  is a ComplexT object
    #  @param c A complex number (ComplexT object)
    #  @return ComplexT object representing the sum of the two complex numbers
```

```
    def add(self, c):
        return ComplexT(c.real() + self.x, c.imag() + self.y)

    ## @brief Subtracts another complex number from the current complex number
    #  @details It is assumed that the input parameter, c,
    #  is a ComplexT object
    #  @param c A complex number (ComplexT object)
    #  @return ComplexT object representing the difference
    #  between the current complex number and the passed complex number
    def sub(self, c):
        return ComplexT(self.x - c.real(), self.y - c.imag())

    ## @brief Multiplies another complex number and the current complex number
    #  @details Complex number multiplication sourced from
    #  https://en.wikipedia.org/wiki/Complex_number
    #  @details It is assumed that the input parameter, c,
    #  is a ComplexT object
    #  @param c A complex number (ComplexT object)
    #  @return ComplexT object representing
    #  the product of the complex numbers
    def mult(self, c):
        # perform binomial expansion for the complex numbers
        # i^2 becomes -1
        x = self.x * c.real() - self.y * c.imag()
        y = self.x * c.imag() + self.y * c.real()
        return ComplexT(x, y)

    ## @brief Gets the reciprocal of the complex number
    #  @details Complex number reciprocal sourced from
    #  https://en.wikipedia.org/wiki/Complex_number
    #  @details It is assumed that this method is
    #  not called when the complex number is 0 + 0i,
    #  due to zero division
    #  @return ComplexT object representing
    #  the reciprocal of the current complex number
    def recip(self):
        # the denominator of both terms is x^2 + y^2
        denom = self.x * self.x + self.y * self.y
        x = self.x / denom
        y = -1 * self.y / denom
        return ComplexT(x, y)

    ## @brief Divides the current complex number by another complex number
    #  @details Complex number division sourced from
    #  https://en.wikipedia.org/wiki/Complex_number
    #  @details It is assumed that the input parameter, c,
    #  is a ComplexT object which is not 0 + 0i, due to zero division
    #  @param c A complex number (ComplexT object)
    #  @return ComplexT object representing
    #  the quotient of the complex numbers
    def div(self, c):
        # division can be rewritten as multiplication of the reciprocal
        return self.mult(c.recip())

    ## @brief Gets the positive square root of the complex number
    #  @details The square root of a complex number sourced from
    #  https://en.wikipedia.org/wiki/Complex_number
    #  @return ComplexT object representing
    #  the positive square root of the complex number
    def sqrt(self):
        c = cmath.sqrt(complex(self.x, self.y))
        return ComplexT(c.real, c.imag)
```

J Code for Partner's triangle_adt.py

```
## @file triangle_adt.py
#  @author Samarth Kumar
#  @brief Contains a class for working with triangles
#  @date 01/21/2021

from math import sqrt
from enum import Enum, auto

## @brief An ADT for handling triangles
```

```
#  @details A triangle consists of its 3 Integer sidelengths
class TriangleT:

    ## @brief Constructor for TriangleT
    #  @details Creates a triangle given its 3 sidelengths
    #  @details It is assumed that sidelengths are passed as Integers
    #  @param x First Integer sidelength
    #  @param y Second Integer sidelength
    #  @param z Third Integer sidelength
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

    ## @brief Gets the sidelengths of the triangle
    #  @return Tuple containing the 3 Integer sidelengths
    def get_sides(self):
        return self.x, self.y, self.z

    ## @brief Check if another triangle is equal to the current triangle object
    #  @details Uses the __eq__() method to determine exact triangle equality
    #  @details Since sidelengths are Integers, they must
    #  exactly match to be equal triangles
    #  @details It is assumed that two invalid triangles can still be equal
    #  @details It is assumed that the input parameter, t,
    #  is a TriangleT object
    #  @param t A triangle (TriangleT object)
    #  @return True if the triangles are equal, False otherwise
    def equal(self, t):
        return self == t

    # Private method for defining TriangleT '==' behaviour
    def __eq__(self, t):
        # Sort both triangle sidelengths first -- uses a version of mergesort
        sides1 = sorted(self.get_sides())
        sides2 = sorted(t.get_sides())
        # '==' will return true iff each index of the tuples match
        return sides1 == sides2

    ## @brief Gets the perimeter of the triangle
    #  @return Integer value of the perimeter of the triangle
    #  or 0 if the triangle is not valid
    def perim(self):
        if not self.is_valid():
            return 0
        return self.x + self.y + self.z

    ## @brief Gets the area of the triangle
    #  @details Area is calculated using Heron's formula as obtained from:
    #  https://en.wikipedia.org/wiki/Heron's_formula
    #  @return Float value of the area of the triangle
    #  or 0.0 if the triangle is not valid
    def area(self):
        if not self.is_valid():
            return 0.0
        s = self.perim() / 2
        return sqrt(s * (s - self.x) * (s - self.y) * (s - self.z))

    ## @brief Checks if the 3 sidelengths of the triangle form a valid triangle
    #  @details For validity, the sum of any 2 sidelengths
    #  must be greater than the third sidelength
    #  @details Reference material obtained from:
    #  https://en.wikipedia.org/wiki/Triangle_inequality
    #  @details It is assumed that a degenerate triangle
    #  (collinear sides and 0 area) is invalid
    #  @return True if it is a valid triangle, False otherwise
    def is_valid(self):
        # Immediately return False if there are sidelengths below or equal to 0
        for s in self.get_sides():
            if s <= 0:
                return False
        # Check that the sum of each pair of sidelengths is greater than
        # the other sidelength
        return ((self.x + self.y > self.z) and (self.x + self.z > self.y) and (self.y + self.z >
            self.x))

    ## @brief Determines the type of triangle
    #  @details A triangle can be classified as equilateral, isosceles,
    #  scalene, or right
    #  @details Sidelengths must exactly match for equality,
```

```
    #  since they are Integers
    #  @details A triangle may meet the criteria for more than one TriType,
    #  but it is classified as only one of them
    #  @details All equilateral triangles are also isosceles,
    #  but the triangle is classified as equilateral
    #  @details If a triangle is both right and scalene,
    #  it is classified as right
    #  @details It is assumed that the current TriangleT object
    #  is a valid triangle
    #  @return TriType which represents the type of triangle
    def tri_type(self):
        # Equilateral
        if self.x == self.y == self.z:
            return TriType.equilat

        # Right
        # Uses Pythagorean theorem to check if it is a right triangle
        s = sorted(self.get_sides())
        if s[0] * s[0] + s[1] * s[1] == s[2] * s[2]:
            return TriType.right

        # Scalene
        if (self.x != self.y) and (self.x != self.z) and (self.y != self.z):
            return TriType.scalene

        # Isosceles
        # All other classifications have been handled, must be isosceles
        return TriType.isosceles

## @brief TriType contains an enumeration for types of triangles
#  @details A triangle can be typed as equilateral, isosceles,
#  scalene, or right
class TriType(Enum):
    equilat = auto()
    isosceles = auto()
    scalene = auto()
    right = auto()
```