Assignment 1 Solution

Rizwan Ahsan, ahsanm7

January 27, 2021

This report discusses testing of the `ComplexT` and `TriangleT` classes written for Assignment 1. It also discusses testing of the partner's version of the two classes. The design restrictions for the assignment are critiqued and then various related discussion questions are answered.

1 Assumptions and Exceptions

The following assumptions were made:

- The arguments provided for the constructor of `ComplexT` class will always be of type float.
- The arguments provided for the constructor of `TriangleT` class will always be of type integer and their values will be greater than 0.

The following exceptions were handled:

- The `recip` method will throw an exception when the value of the denominator variable calculated from the object's real and imaginary numbers is less than or equal to 0 i.e $x^2 + y^2 <= 0$, x being the real part and y being the imaginary part.
- The `div` method will throw a similar exception to `recip` but in this case the exception is thrown for the passed in argument's real and imaginary numbers instead.

2 Test Cases and Rationale

All of the test cases that were written covers all of the methods in `ComplexT` and `TriangleT`. The test cases covers all branches of the code meaning all control flow statements were checked i.e all possible outcomes from if-else were covered. The part of the code that

raised exceptions were also handled specifically. Some methods had edge cases which were also handled. Otherwise, the rest of the tests that had no weird input issues were handled only once, since that covered all outcomes. On some of the methods that returned floating value as the output, those cases were tested by rounding the output to 5 decimal places. By doing so the test will only return passed if the output matches the desired result upto those 5 decimal places, which in turn reduces the floating point error to some degree.

3 Results of Testing Partner's Code

My tests file had a total of 26 tests. Running my test cases on my partner's files resulted in 25 passes and 1 failure. The one failed case occured in the `div` method of class `ComplexT`. This failure occured due to the way my test case was implemented. The test will only return true only if the output is equal to the calculated output upto 5 decimal places. But since my test case didn't consider for relative floating point error this returned as failed.

4 Critique of Given Design Specification

The given design specification is very good and detailed enough to create the desired outcome. However, it didn't mention everything possible cases properly and left it ambigous for the developer to decide. For example, the specification didn't mention the outcome for the possiblity of giving a wrong input to any of the classes. In my opinion, the design would be more robust to say that it will take in only a specific type of input and handle the remaining inputs somehow. Also, for the `tir_type` method, the design specification didn't mention what would be returned as the output when the provided input corresponds to more than one type of triangle. So, the less the design specification contains ambiguity regarding the functionality the more robust the design specification will become. This will cause the outcome to be more stable.

5 Answers to Questions

(a) There aren't any mutators in `ComplexT` and `TriangleT`.

The methods which are selectors (getters) are:

- `real`
- `imag`
- `get_sides`

(b) The two possible state variables for `ComplexT` are:

- Consider the provided input as the real and imaginary part of the complex number.
- Consider the provided input and set modulus and argument as the state variables.

The two possible state variables for `TriangleT` are:

- Set three different variables as the three sides of the triangle.
- Setting only variable which consists all three sides in list or tuple.

(c) No, I don't think it would make sense to add methods for greater than and less than because whether a triangle is greater or smaller than another has no use in the real world. On the other hand, figuring out if two triangles are equal can help to prove many mathematical theorems.

(d) Yes, it is possible that the three integer inputs to the constructor for `TriangleT` will not form a geometrically valid triangle. This will happen when the lengths of any of the two sides is not greater than the third side. In this case the contructor should initially check for the validity of the inputs and raise an error if they are invalid. By doing so the user will realise that the inputs they have provided is wrong and fix it immediately rather than finding it out later in more complex calculations.

(e) Yes, having the type of the triangle as a state variable is a good idea. This will immediatly allow the user to figure out what type of triangle it is and use all the properties of that specific triangle in their code.

(f) The software qualities of performance and usability has an inverse relationship. If the performance of a product is poor then the product becomes difficult to use and vise versa. For example, if the performance of a phone is poor, then it will take a considerable amount of time for it to complete the tasks that it's user would like to do. This will result in the user not wanting to use that product again.

(g) No, I don't think there are any situations where it is not necessary to fake a rational design process. Even if the entire design process is known prior to the starting of designing software there would still be some unknown edge cases that will only come out during the coding phase and testing it out.

(h) The reusability of code can affect positively towards the reliability of products. For example, it would be unwise to create something from the start if something that

works already exists. Like while creating a game, creating all the physics of the game would take a long amount of energy and time but instead using game engines that already has it built-in would save not only a lot of time but increase the chances of creating a well polished game.

(i) The idea of abstraction is that it contains three main properties.

- The concept of hiding or removing.
- The concept of generalization.
- The concept of idea versus reality.

Some examples of programming languages being abstractions built on top of hardware are:

- The usage of various data types to perform data abstraction.
- The creation of abstract data types for example, ComplexT, TriangleT to mimic their properties.
- The concept of functions is an abstraction, since the internal mechanism remains hidden to the user. A function is used to generalize a behaviour and it can be reused many times.

F Code for complex_adt.py

```
## @file complex_adt.py
#  @author Rizwan Ahsan
#  @brief Contains a class for working with complex numbers
#  @date 01/20/2021

import math

## @brief An ADT class for representing complex numbers
#  @details A complex number is composed of a real part and an imaginary part
class ComplexT():

    ## @brief Contructor for ComplexT class
    #  @details Creates a complex number based on the provided arguments.
    #  Assumed the arguments provided are going to be of type float
    #  @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
    def real(self):
        return self.__x

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

    ## @brief Gets the modulus of the complex number
    #  @return Float representing the modulus
    def get_r(self):
        return math.sqrt(self.__x ** 2 + self.__y ** 2)

    ## @brief Gets the argument of the complex number
    #  @return Float representing the argument
    def get_phi(self):
        if self.__x > 0:
            return math.atan(self.__y / self.__x)
        elif self.__x < 0:
            if self.__y < 0:
                return math.atan(self.__y / self.__x) - math.pi
            else:
                return math.atan(self.__y / self.__x) + math.pi
        else:
            if self.__y > 0:
                return math.pi / 2
            elif self.__y < 0:
                return - math.pi / 2
            else:
                return 0.0

    ## @brief Checks if the provided complex number is equal to this complex number
    #  @param complex_num ComplexT to check if it is equal to this one
    #  @return True if they are equal else False
    def equal(self, complex_num):
        return self.real() == complex_num.real() and self.imag() == complex_num.imag()

    ## @brief Checks if the provided complex number is equal to this complex number
    #  @param other ComplexT to check if it is equal to this one
    #  @return True if they are equal else False
    def __eq__(self, other):
        return self.equal(other)

    ## @brief Gets the complex conjugate of the complex number
    #  @return ComplexT representing the complex conjugate
    def conj(self):
        return ComplexT(self.__x, - self.__y)

    ## @brief Adds the given complex number to this number
    #  @param complex_num ComplexT number to add to this number
    #  @return ComplexT which results from adding both the numbers
    def add(self, complex_num):
        return ComplexT(self.__x + complex_num.real(), self.__y + complex_num.imag())
```

```
## @brief Subtracts the given complex number from this number
#  @param complex_num ComplexT number to subtract from this number
#  @return ComplexT which results from the subtraction
def sub(self, complex_num):
    return ComplexT(self.__x - complex_num.real(), self.__y - complex_num.imag())

## @brief Multiplies the given complex number and this number
#  @param complex_num ComplexT number to multiply to this number
#  @return ComplexT which results from the multiplication
def mult(self, complex_num):
    x = self.__x * complex_num.real() - self.__y * complex_num.imag()
    y = self.__x * complex_num.imag() + self.__y * complex_num.real()
    return ComplexT(x, y)

## @brief Gets the reciprocal of the complex number
#  @return ComplexT which is the reciprocal of this complex number
#  @throws Exception throws if the denom variable is less than or equal to 0
def recip(self):
    denom = self.__x ** 2 + self.__y ** 2
    if denom <= 0:
        raise Exception("The denominator should be non-zero i.e. x^2 + y^2 must be greater than 0")
    return ComplexT(self.__x / denom, - self.__y / denom)

## @brief Divides this number by the provided complex number
#  @param complex_num ComplexT number which is used to divide this number
#  @return ComplexT which results from the division of the numbers
#  @throws Exception throws if the denom variable is less than or equal to 0
def div(self, complex_num):
    denom = complex_num.real() ** 2 + complex_num.imag() ** 2
    if denom <= 0:
        raise Exception("The argument must be a non-zero complex number")
    x = self.__x * complex_num.real() + self.__y * complex_num.imag()
    y = self.__y * complex_num.real() - self.__x * complex_num.imag()
    return ComplexT(x / denom, y / denom)

## @brief Gets the positive square root of the complex number
#  @return ComplexT which is the positive square root of the complex number
def sqrt(self):
    r = self.get_r()
    theta = self.get_phi()
    x = math.sqrt(r) * math.cos(theta / 2)
    y = math.sqrt(r) * math.sin(theta / 2)
    return ComplexT(x, y)
```

G Code for triangle_adt.py

```
## @file triangle_adt.py
#  @author Rizwan Ahsan
#  @brief Contains a class for working with triangles
#  @date 01/20/2021

import math
from enum import Enum, auto

## @brief An ADT class for representing triangles
#  @details A triangle is composed of three sides
class TriangleT():

    ## @brief Contructor for TriangleT class
    #  @details Creates a triangle based on the provided arguments. Assumed the sides are
    #  of type integer and greater than 0
    #  @param x Integer representing a side of the triangle
    #  @param y Integer representing a side of the triangle
    #  @param z Integer representing a side of the triangle
    def __init__(self, x, y, z):
        self.__x = x
        self.__y = y
        self.__z = z

    ## @brief Gets the values of all the sides of the triangle
    #  @return Tuple containing all the sides
    def get_sides(self):
        return (self.__x, self.__y, self.__z)

    ## @brief Checks if the provided triangle is equal to this triangle
    #  @param triangle The triangle which is used to compare the equality
    #  @return True is both the triangles are equal, false otherwise
    def equal(self, triangle):
        sides1 = self.get_sides()
        sides2 = triangle.get_sides()
        return sides1[0] == sides2[0] and sides1[1] == sides2[1] and sides1[2] == sides2[2]

    ## @brief Gets the perimeter of the triangle
    #  @return Integer which represents the perimeter of the triangle
    def perim(self):
        return self.__z + self.__y + self.__x

    ## @brief Gets the area of the traingle
    #  @return Float representing the area of the triangle
    def area(self):
        half = self.perim() / 2
        return math.sqrt(half * (half - self.__x) * (half - self.__y) * (half - self.__z))

    ## @brief Checks if the provided sides creates a valid triangle or not
    #  @return True if the triangle is valid, false otherwise
    def is_valid(self):
        cond1 = self.__x + self.__y <= self.__z
        cond2 = self.__z + self.__y <= self.__x
        cond3 = self.__x + self.__z <= self.__y
        if cond1 or cond2 or cond3:
            return False
        else:
            return True

    ## @brief Gets a TriType by checking with the provided sides
    #  @return TriType for the provided sides
    def tri_type(self):
        if self.__y == self.__x == self.__z:
            return TriType.equilat

        if self.__x == self.__y or self.__y == self.__z or self.__z == self.__x:
            return TriType.isosceles

        a, b, c = self.__x ** 2, self.__y ** 2, self.__z ** 2
        if (a + b + c - max(a, b, c)) == max(a, b, c):
            return TriType.right

        if self.__y != self.__x != self.__z:
            return TriType.scalene

## @brief TriType class is an enumeration of the different types of triangle
#  @details The four different types of triangle are: equilateral, isosceles,
```

```
#  scalene and right angled triangle
class TriType(Enum):

    ## enum value equilat
    equilat = auto()

    ## enum value isosceles
    isosceles = auto()

    ## enum value scalene
    scalene = auto()

    ## enum value right
    right = auto()
```

H Code for test_driver.py

```
## @file test_driver.py
#  @author Rizwan Ahsan
#  @brief Tests for complex_adt.py and triangle_adt.py
#  @date 01/20/2021

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

passed = 0
failed = 0

##complex_adt.py tests
a = ComplexT(2.0, 5.0)
b = complex(2.0, 5.0)
c = ComplexT(1.0, 3.0)
f = complex(1.0, 3.0)
d = b.__sub__(f)
e = b.__mul__(f)

#test of real method
if a.real() == b.real:
    print("ComplexT.real() test PASSES")
    passed += 1
else:
    print("ComplexT.real() test FAILS")
    failed += 1

#test of imag method
if a.imag() == b.imag:
    print("ComplexT.imag() test PASSES")
    passed += 1
else:
    print("ComplexT.imag() test FAILS")
    failed += 1

#test of get_r method
if a.get_r() == abs(b):
    print("ComplexT.get_r() test PASSES")
    passed += 1
else:
    print("ComplexT.get_r() test FAILS")
    failed += 1

#test of get_phi method
#test 1
if a.get_phi() == cmath.phase(b):
    print("ComplexT.get_phi() test 1 PASSES")
    passed += 1
else:
    print("ComplexT.get_phi() test 1 FAILS")
    failed += 1

#test 2
if ComplexT(-10.0, 0.0).get_phi() == cmath.phase(complex(-10.0, 0.0)):
    print("ComplexT.get_phi() test 2 PASSES")
    passed += 1
else:
    print("ComplexT.get_phi() test 2 FAILS")
    failed += 1

#test 3
if ComplexT(0.0, 0.0).get_phi() == 0.0:
    print("ComplexT.get_phi() test 3 PASSES")
    passed += 1
else:
    print("ComplexT.get_phi() test 3 FAILS")
    failed += 1

#test 4
if ComplexT(0.0, 50.0).get_phi() == cmath.phase(complex(0.0, 50.0)):
    print("ComplexT.get_phi() test 4 PASSES")
    passed += 1
else:
    print("ComplexT.get_phi() test 4 FAILS")
    failed += 1
```

```
#test of equal method
if not a.equal(c):
    print("ComplexT.equal() test PASSES")
    passed += 1
else:
    print("ComplexT.equal() test FAILS")
    failed += 1

#test of conj method
if a.conj() == ComplexT(b.conjugate().real, b.conjugate().imag):
    print("ComplexT.conj() test PASSES")
    passed += 1
else:
    print("ComplexT.conj() test FAILS")
    failed += 1

#test of add method
if a.add(c) == ComplexT(3.0, 8.0):
    print("ComplexT.add() test PASSES")
    passed += 1
else:
    print("ComplexT.add() test FAILS")
    failed += 1

#test of sub method
if a.sub(c) == ComplexT(d.real, d.imag):
    print("ComplexT.sub() test PASSES")
    passed += 1
else:
    print("ComplexT.sub() test FAILS")
    failed += 1

#test of mult method
if a.mult(c) == ComplexT(e.real, e.imag):
    print("ComplexT.mult() test PASSES")
    passed += 1
else:
    print("ComplexT.mult() test FAILS")
    failed += 1

#test for recip method
if a.recip() == ComplexT(2/29, -5/29):
    print("ComplexT.recip() test PASSES")
    passed += 1
else:
    print("ComplexT.recip() test FAILS")
    failed += 1

#special test for recip method
try:
    ComplexT(0.0, 0.0).recip()
    print("ComplexT.recip() special test FAILS")
    failed += 1
except:
    print("ComplexT.recip() special test PASSES")
    passed += 1

#test for div method
div1 = a.div(c)
div2 = b.__truediv__(f)
if div1 == ComplexT(round(div2.real, 5), round(div2.imag, 5)):
    print("ComplexT.div() test PASSES")
    passed += 1
else:
    print("ComplexT.div() test FAILS")
    failed += 1

#special test for div method
try:
    a.div(ComplexT(0.0, 0.0))
    print("ComplexT.div() special test FAILS")
    failed += 1
except:
    print("ComplexT.div() special test PASSES")
    passed += 1

#test for sqrt method
sqrt = a.sqrt()
sqrt2 = cmath.sqrt(b)
```

```
if ComplexT(round(sqrt.real(), 5), round(sqrt.imag(), 5)) == ComplexT(round(sqrt2.real, 5),
     round(sqrt2.imag, 5)):
     print("ComplexT.sqrt() test PASSES")
     passed += 1
else:
     print("ComplexT.sqrt() test FAILS")
     failed += 1


##triangle_adt.py tests
tri = TriangleT(3, 4, 5)
tri2 = TriangleT(5, 6, 7)

#test for get_sides method
if tri.get_sides() == (3, 4, 5):
     print("TriangleT.get_sides() test PASSES")
     passed += 1
else:
     print("TriangleT.get_sides() test FAILS")
     failed += 1

#test for equal method
if not tri.equal(tri2):
     print("TriangleT.equal() test PASSES")
     passed += 1
else:
     print("TriangleT.equal() test FAILS")
     failed += 1

#test for get_sides method
if tri.perim() == 12:
     print("TriangleT.perim() test PASSES")
     passed += 1
else:
     print("TriangleT.perim() test FAILS")
     failed += 1

#test for area method
if tri.area() == 6:
     print("TriangleT.area() test PASSES")
     passed += 1
else:
     print("TriangleT.area() test FAILS")
     failed += 1

#test for is_valid method
if tri.is_valid():
     print("TriangleT.is_valid() test PASSES")
     passed += 1
else:
     print("TriangleT.is_valid() test FAILS")
     failed += 1

#test for tri_type method
#test 1
if tri.tri_type() == TriType.right:
     print("TriangleT.tri_type() test 1 PASSES")
     passed += 1
else:
     print("TriangleT.tri_type() test 1 FAILS")
     failed += 1

#test 2
if TriangleT(5,5,5).tri_type() == TriType.equilat:
     print("TriangleT.tri_type() test 2 PASSES")
     passed += 1
else:
     print("TriangleT.tri_type() test 2 FAILS")
     failed += 1

#test 3
if TriangleT(7,5,5).tri_type() == TriType.isosceles:
     print("TriangleT.tri_type() test 3 PASSES")
     passed += 1
else:
     print("TriangleT.tri_type() test 3 FAILS")
     failed += 1

#test 4
if TriangleT(7,9,13).tri_type() == TriType.scalene:
```

```
    print("TriangleT.tri_type() test 4 PASSES")
    passed += 1
else:
    print("TriangleT.tri_type() test 4 FAILS")
    failed += 1

print(f"\nPASSED: {passed}/26, FAILED: {failed}/26")
```

I Code for Partner's complex_adt.py

```
## @file complex_adt.py
#  @author Emily Sanderson
#  @brief Provides the ComplexT ADT class for representing complex numbers
#  @date Jan 18/2021

from math import sqrt
import numpy as np


## @brief An ADT that represents a complex number
class ComplexT:

    ## @brief ComplexT constructor
    #  @details Initializes a ComplexT obkect with a real & imaginary part
    #  @param x x is the real part of the complex number
    #  @param y y is the imaginary part of the complex number
    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    ## @brief Compare two ComplexT Objects
    #  @param other Other ComplexT Object to be compared to
    #  @return Returns True if objs have the same real & imag parts, False o/w
    def __eq__(self, other):
        if (other.__x == self.__x and other.__y == self.__y):
            return True
        return False

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

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

    ## @brief Calculates the abs value of complex number
    #  @return  The absolute value of the complex number
    def get_r(self):
        return sqrt(self.__x ** 2 + self.__y ** 2)

    ## @brief Calculates the argument of the complex number
    #  @return The argument of the complex number
    def get_phi(self):
        return np.arctan2(self.__y, self.__x)

    ## @brief Determines if the argument and current object are equal
    #  @param obj obj is a ComplexT object used for comparison
    #  @return Returns true if arg and current obj are equal
    def equal(self, obj):
        return self.__eq__(obj)

    ## @brief Calculates the complex conjugate
    #  @return Returns a ComplexT object with new params
    def conj(self):
        return ComplexT(self.__x, (-1) * self.__y)

    ## @brief Calculates the addition of two ComplexT objects
    #  @param obj ComplexT object being added to self
    #  @return Returns a ComplexT object that is the addition of obj and self
    def add(self, obj):
        return ComplexT(self.__x + obj.__x, self.__y + obj.__y)

    ## @brief Calculates the subtraction of the ComplexT argument from obj
    #  @param obj ComplexT object being subtracted from self
    #  @return ComplexT object that is the subtraction of obj minus the arg
    def sub(self, obj):
        return ComplexT(self.__x - obj.__x, self.__y - obj.__y)

    ## @brief Calculates the multiplication of two ComplexT objects
    #  @param obj ComplexT object that is to be multipled by self
    #  @return ComplexT object that is the mult of current obj and param
    def mult(self, obj):
        return ComplexT(self.__x * obj.__x - self.__y * obj.__y,
```

```
                        self.__x * obj.__y + self.__y * obj.__x)

    ## @brief Calculates the reciprocal of ComplexT object
    #  @throws ValueError if Complex Number is 0
    #  @return Returns the Reciprocal of the complex number
    def recip(self):
        if (self.__x == 0 and self.__y == 0):
            raise ValueError("Cannot divide by 0")
        return ComplexT(self.__x / (self.__x ** 2 + self.__y ** 2),
                        -self.__y / (self.__x ** 2 + self.__y ** 2))

    ## @brief Calculates the division of two ComplexT objects
    #  @param obj ComplexT object that divides self
    #  @throws ValueError if denominator Complex Number is 0
    #  @return ComplexT object that is the results of the div of current obj and param
    def div(self, obj):
        if (obj.__x == 0 and obj.__y == 0):
            raise ValueError("Cannot divide by 0")
        return self.mult(obj.recip())

    ## @brief Calculates the positive square root of the current object
    #  @return Returns the positive square root of the complex number
    def sqrt(self):
        if (self.__y == 0):
            return ComplexT(sqrt(self.__x), 0)
        else:
            return ComplexT(sqrt((self.get_r() + self.__x) / 2),
                            (self.__y / abs(self.__y)) *
                            sqrt((self.get_r() - self.__x) / 2))
```

J Code for Partner's triangle_adt.py

```
## @file triangle_adt.py
#  @author Emily Sanderson
#  @brief Provides the TriangleT ADT class for representing a Triangle
#  @date Jan 18/2021

from math import sqrt
from enum import Enum
from itertools import permutations


## @brief An ADT that represents a triangle
class TriangleT:

    ## @brief TriangleT constructor
    #  @details Initializes a TriangleT object
    #  @throws ValueError if triangle has negative side lengths
    #  @param a Length of first side of Triangle
    #  @param b Length of second side of Triangle
    #  @param c Length of third side of Triangle
    def __init__(self, a, b, c):
        if (a < 0 or b < 0 or c < 0):
            raise ValueError("Triangle cannot have negative side lengths")
        self.__a = a
        self.__b = b
        self.__c = c

    ## @brief Gets the sides of a triangle
    #  @return Returns a tuple of three integers
    def get_sides(self):
        return (self.__a, self.__b, self.__c)

    ## @brief Sorts sides of triangle
    #  @return Returns an array of sorted sides lengths
    def __sort_sides__(self):
        return sorted(self.get_sides())

    ## @brief Tests whether two triangles are equivalent
    #  @param obj Other triangle used in comparison
    #  @return Returns True if two triangleT objects are the same
    def equal(self, obj):
        self_sides = self.__sort_sides__()
        obj_sides = obj.__sort_sides__()
```

```
        if (self_sides == obj_sides):
            return True
        return False

    ## @brief Calculates the perimeter of Triangle
    #  @throws ValueError if triangle is invalid
    #  @return Returns the perimeter as an int of Triangle
    def perim(self):
        if not self.is_valid():
            raise Exception("Invalid Triangle")
        return self.__a + self.__b + self.__c

    ## @brief Calculates the area of a triangle
    #  @throws ValueError if triangle is invalid
    #  @return Returns the area of a triangle, as a float
    def area(self):
        if not self.is_valid():
            raise Exception("Invalid Triangle")
        p = self.perim() / 2
        return sqrt(p * (p - self.__a) * (p - self.__b) * (p - self.__c))

    ## @brief Determines is the triangleT object represents a valid Triangle
    #  @details The sum of two sides must be greater than the third side to be valid
    #  @return Returns True is triangleT rep a valid triangle, False o/w
    def is_valid(self):
        if (self.__a + self.__b > self.__c and
                self.__a + self.__c > self.__b and
                self.__c + self.__b > self.__a):
            return True
        return False

    ## @brief Creates an object of the Class TriType
    #  @throws ValueError if triangle is invalid
    #  @return Returns a TriType object
    def tri_type(self):
        if not self.is_valid():
            raise Exception("Invalid Triangle")
        sorted_sides = self.__sort_sides__()
        for perm in permutations(self.get_sides()):
            if (perm[0]**2 + perm[1]**2 == perm[2]**2):
                return TriType.right
        if (sorted_sides[0] == sorted_sides[2]):
            return TriType.equilat
        elif ((sorted_sides[0] == sorted_sides[1]) or
              (sorted_sides[1] == sorted_sides[2])):
            return TriType.isosceles
        else:
            return TriType.scalene


## @brief An Enum that represents the types of Triangles
#  @details Elements of TriType correspond to the triangle types
class TriType(Enum):
    equilat = 1
    isosceles = 2
    scalene = 3
    right = 4
```